Valgrind Documentation

Release 3.25.1 20 May 2025

Copyright © 2000-2025 AUTHORS

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled The GNU Free Documentation License.

This is the top level of Valgrind's documentation tree. The documentation is contained in six logically separate documents, as listed in the following Table of Contents. To get started quickly, read the Valgrind Quick Start Guide. For full documentation on Valgrind, read the Valgrind User Manual.

Table of Contents

The Valgrind Quick Start Guide

Release 3.25.1 20 May 2025

Copyright © 2000-2025 Valgrind Developers

Email: valgrind@valgrind.org

Table of Contents

The Valgrind Quick Start Guide

1. Introduction

The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour.

The rest of this guide gives the minimum information you need to start detecting memory errors in your program with Memcheck. For full documentation of Memcheck and the other tools, please read the User Manual.

2. Preparing your program

Compile your program with `-g` to include debugging information so that Memcheck's error messages include exact line numbers. Using `-O0` is also a good idea, if you can tolerate the slowdown. With `-O1` line numbers in error messages can be inaccurate, although generally speaking running Memcheck on code compiled at `-O1` works fairly well, and the speed improvement compared to running `-O0` is quite significant. Use of `-O2` and above is not recommended as Memcheck occasionally reports uninitialised-value errors which don't really exist.

3. Running your program under Memcheck

If you normally run your program like this:

```
myprog arg1 arg2
```

Use this command line:

```
valgrind --leak-check=yes myprog arg1 arg2
```

Memcheck is the default tool. The `--leak-check` option turns on the detailed memory leak detector.

Your program will run much slower (eg. 20 to 30 times) than normal, and use a lot more memory. Memcheck will issue messages about memory errors and leaks that it detects.

4. Interpreting Memcheck's output

Here's an example C program, in a file called a.c, with a memory error and a memory leak.

```
#include <stdlib.h>

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
}                    // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
```

```
}
```

Most error messages look like the following, which describes problem 1, the heap block overrun:

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Things to notice:

- There is a lot of information in each error message; read it carefully.
- The 19182 is the process ID; it's usually unimportant.
- The first line ("Invalid write...") tells you what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.
- Below the first line is a stack trace telling you where the problem occurred. Stack traces can get quite large, and be confusing, especially if you are using the C++ STL. Reading them from the bottom up can help. If the stack trace is not big enough, use the `--num-callers` option to make it bigger.
- The code addresses (eg. 0x804838F) are usually unimportant, but occasionally crucial for tracking down weirder bugs.
- Some error messages have a second component which describes the memory address involved. This one shows that the written memory is just past the end of a block allocated with malloc() on line 5 of example.c.

It's worth fixing errors in the order they are reported, as later errors can be caused by earlier errors. Failing to do this is a common cause of difficulty with Memcheck.

Memory leak messages look like this:

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

The stack trace tells you where the leaked memory was allocated. Memcheck cannot tell you why the memory leaked, unfortunately. (Ignore the "vg_replace_malloc.c", that's an implementation detail.)

There are several kinds of leaks; the two most important categories are:

- "definitely lost": your program is leaking memory -- fix it!
- "probably lost": your program is leaking memory, unless you're doing funny things with pointers (such as moving them to point to the middle of a heap block).

Memcheck also reports uses of uninitialised values, most commonly with the message "Conditional jump or move depends on uninitialised value(s)". It can be difficult to determine the root cause of these errors. Try using the `--track-origins=yes` to get extra information. This makes Memcheck run slower, but the extra information you get often saves a lot of time figuring out where the uninitialised values are coming from.

If you don't understand an error message, please consult Explanation of error messages from Memcheck in the Valgrind User Manual which has examples of all the error messages Memcheck produces.

5. Caveats

Memcheck is not perfect; it occasionally produces false positives, and there are mechanisms for suppressing these (see Suppressing errors in the Valgrind User Manual). However, it is typically right 99% of the time, so you should be wary of ignoring its error messages. After all, you wouldn't ignore warning messages produced by a compiler, right? The suppression mechanism is also useful if Memcheck is reporting errors in library code that you cannot change. The default suppression set hides a lot of these, but you may come across more.

Memcheck cannot detect every memory error your program has. For example, it can't detect out-of-range reads or writes to arrays that are allocated statically or on the stack. But it should detect many errors that could crash your program (eg. cause a segmentation fault).

Try to make your program so clean that Memcheck reports no errors. Once you achieve this state, it is much easier to see when changes to the program cause Memcheck to report new errors. Experience from several years of Memcheck use shows that it is possible to make even huge programs run Memcheck-clean. For example, large parts of KDE, OpenOffice.org and Firefox are Memcheck-clean, or very close to it.

6. More information

Please consult the Valgrind FAQ and the Valgrind User Manual, which have much more information. Note that the other tools in the Valgrind distribution can be invoked with the `--tool` option.

Valgrind User Manual

Release 3.25.1 20 May 2025

Copyright © 2000-2025 Valgrind Developers

Email: valgrind@valgrind.org

Table of Contents

1. Introduction

1.1. An Overview of Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools. It comes with a set of tools each of which performs some kind of debugging, profiling, or similar task that helps you improve your programs. Valgrind's architecture is modular, so new tools can be created easily and without disturbing the existing structure.

A number of useful tools are supplied as standard.

1. **Memcheck** is a memory error detector. It helps you make your programs, particularly those written in C and C++, more correct.
2. **Cachegrind** is a cache and branch-prediction profiler. It helps you make your programs run faster.
3. **Callgrind** is a call-graph generating cache profiler. It has some overlap with Cachegrind, but also gathers some information that Cachegrind does not.
4. **Helgrind** is a thread error detector. It helps you make your multi-threaded programs more correct.
5. **DRD** is also a thread error detector. It is similar to Helgrind but uses different analysis techniques and so may find different problems.
6. **Massif** is a heap profiler. It helps you make your programs use less memory.
7. **DHAT** is a different kind of heap profiler. It helps you understand issues of block lifetimes, block utilisation, and layout inefficiencies.
8. **BBV** is an experimental SimPoint basic block vector generator. It is useful to people doing computer architecture research and development.

There are also a couple of minor tools that aren't useful to most users: **Lackey** is an example tool that illustrates some instrumentation basics; and **Nulgrind** is the minimal Valgrind tool that does no analysis or instrumentation, and is only useful for testing purposes.

Valgrind is closely tied to details of the CPU and operating system, and to a lesser extent, the compiler and basic C libraries. Nonetheless, it supports a number of widely-used platforms, listed in full at http://www.valgrind.org/.

Valgrind is built via the standard Unix `./configure`, `make`, `make install` process; full details are given in the README file in the distribution.

Valgrind is licensed under the The GNU General Public License, version 2. The `valgrind/*.h` headers that you may wish to include in your code (eg. `valgrind.h`, `memcheck.h`, `helgrind.h`, etc.) are distributed under a BSD-style license, so you may include them in your code without worrying about license conflicts. Some of the PThreads test cases, `pth_*.c`, are taken from "Pthreads Programming" by Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrell, ISBN 1-56592-115-1, published by O'Reilly & Associates, Inc.

If you contribute code to Valgrind, please ensure your contributions are licensed as "GPLv2, or (at your option) any later version." This is so as to allow the possibility of easily upgrading the license to GPLv3 in future. If you want to modify code in the VEX subdirectory, please also see the file VEX/HACKING.README in the distribution.

1.2. How to navigate this manual

This manual's structure reflects the structure of Valgrind itself. First, we describe the Valgrind core, how to use it, and the options it supports. Then, each tool has its own chapter in this manual. You only need to read the documentation for the core and for the tool(s) you actually use, although you may find it helpful to be at least a little bit familiar with what all tools do. If you're new to all this, you probably want to run the Memcheck tool and you might find the The Valgrind Quick Start Guide useful.

Be aware that the core understands some command line options, and the tools have their own options which they know about. This means there is no central place describing all the options that are accepted -- you have to read the options documentation both for Valgrind's core and for the tool you want to use.

2. Using and understanding the Valgrind core

This chapter describes the Valgrind core services, command-line options and behaviours. That means it is relevant regardless of what particular tool you are using. The information should be sufficient for you to make effective day-to-day use of Valgrind. Advanced topics related to the Valgrind core are described in Valgrind's core: advanced topics.

A point of terminology: most references to "Valgrind" in this chapter refer to the Valgrind core services.

2.1. What Valgrind does with your program

Valgrind is designed to be as non-intrusive as possible. It works directly with existing executables. You don't need to recompile, relink, or otherwise modify the program to be checked.

You invoke Valgrind like this:

```
valgrind [valgrind-options] your-prog [your-prog-options]
```

The most important option is `--tool` which dictates which Valgrind tool to run. For example, if want to run the command `ls -l` using the memory-checking tool Memcheck, issue this command:

```
valgrind --tool=memcheck ls -l
```

However, Memcheck is the default, so if you want to use it you can omit the `--tool` option.

Regardless of which tool is in use, Valgrind takes control of your program before it starts. Debugging information is read from the executable and associated libraries, so that error messages and other outputs can be phrased in terms of source code locations, when appropriate.

Your program is then run on a synthetic CPU provided by the Valgrind core. As new code is executed for the first time, the core hands the code to the selected tool. The tool adds its own instrumentation code to this and hands the result back to the core, which coordinates the continued execution of this instrumented code.

The amount of instrumentation code added varies widely between tools. At one end of the scale, Memcheck adds code to check every memory access and every value computed, making it run 10-50 times slower than natively. At the other end of the spectrum, the minimal tool, called Nulgrind, adds no instrumentation at all and causes in total "only" about a 4 times slowdown.

Valgrind simulates every single instruction your program executes. Because of this, the active tool checks, or profiles, not only the code in your application but also in all supporting dynamically-linked libraries, including the C library, graphical libraries, and so on.

If you're using an error-detection tool, Valgrind may detect errors in system libraries, for example the GNU C or X11 libraries, which you have to use. You might not be interested in these errors, since you probably have no control over that code. Therefore, Valgrind allows you to selectively suppress errors, by recording them in a suppressions file which is read when Valgrind starts up. The build mechanism selects default suppressions which give reasonable behaviour for the OS and libraries detected on your machine. To make it easier to write suppressions, you can use the `--gen-suppressions=yes` option. This tells Valgrind to print out a suppression for each reported error, which you can then copy into a suppressions file.

Valgrind will try to match the behaviour of applications compiled to run on the same OS and libraries that Valgrind was built with. If you use different libraries or a different OS version there may be some small differences in behaviour.

Different error-checking tools report different kinds of errors. The suppression mechanism therefore allows you to say which tool or tool(s) each suppression applies to.

2.2. Getting started

First off, consider whether it might be beneficial to recompile your application and supporting libraries with debugging info enabled (the `-g` option). Without debugging info, the best Valgrind tools will be able to do is guess which function a particular piece of code belongs to, which makes both error messages and profiling output nearly useless. With `-g`, you'll get messages which point directly to the relevant source code lines.

Another option you might like to consider, if you are working with C++, is `-fno-inline`. That makes it easier to see the function-call chain, which can help reduce confusion when navigating around large C++ apps. For example, debugging OpenOffice.org with Memcheck is a bit easier when using this option. You don't have to do this, but doing so helps Valgrind produce more accurate and less confusing error reports. Chances are you're set up like this already, if you intended to debug your program with GNU GDB, or some other debugger. Alternatively, the Valgrind option `--read-inline-info=yes` instructs Valgrind to read the debug information describing inlining information. With this, function call chain will be properly shown, even when your application is compiled with inlining.

If you are planning to use Memcheck: On rare occasions, compiler optimisations (at `-O2` and above, and sometimes `-O1`) have been observed to generate code which fools Memcheck into wrongly reporting uninitialised value errors, or missing uninitialised value errors. We have looked in detail into fixing this, and unfortunately the result is that doing so would give a further significant slowdown in what is already a slow tool. So the best solution is to turn off optimisation altogether. Since this often makes things unmanageably slow, a reasonable compromise is to use `-O`. This gets you the majority of the benefits of higher optimisation levels whilst keeping relatively small the chances of false positives or false negatives from Memcheck. Also, you should compile your code with `-Wall` because it can identify some or all of the problems that Valgrind can miss at the higher optimisation levels. (Using `-Wall` is also a good idea in general.) All other tools (as far as we know) are unaffected by optimisation level, and for profiling tools like Cachegrind it is better to compile your program at its normal optimisation level.

Valgrind understands the DWARF2/3/4 formats used by GCC 3.1 and later. The reader for "stabs" debugging format (used by GCC versions prior to 3.1) has been disabled in Valgrind 3.9.0.

When you're ready to roll, run Valgrind as described above. Note that you should run the real (machine-code) executable here. If your application is started by, for example, a shell or Perl script, you'll need to modify it to invoke Valgrind on the real executables. Running such scripts directly under Valgrind will result in you getting error reports pertaining to `/bin/sh`, `/usr/bin/perl`, or whatever interpreter you're using. This may not be what you want and can be confusing. You can force the issue by giving the option `--trace-children=yes`, but confusion is still likely.

2.3. The Commentary

Valgrind tools write a commentary, a stream of text, detailing error reports and other significant events. All lines in the commentary have following form:

```
==12345== some-message-from-Valgrind
```

The `12345` is the process ID. This scheme makes it easy to distinguish program output from Valgrind commentary, and also easy to differentiate commentaries from different processes which have become merged together, for whatever reason.

By default, Valgrind tools write only essential messages to the commentary, so as to avoid flooding you with information of secondary importance. If you want more information about what is happening, re-run, passing the `-v` option to Valgrind. A second `-v` gives yet more detail.

You can direct the commentary to three different places:

1. The default: send it to a file descriptor, which is by default 2 (stderr). So, if you give the core no options, it will write commentary to the standard error stream. If you want to send it to some other file descriptor, for example number 9, you can specify `--log-fd=9`.

   This is the simplest and most common arrangement, but can cause problems when Valgrinding entire trees of processes which expect specific file descriptors, particularly stdin/stdout/stderr, to be available for their own use.

2. A less intrusive option is to write the commentary to a file, which you specify by `--log-file=filename`. There are special format specifiers that can be used to use a process ID or an environment variable name in the log file name. These are useful/necessary if your program invokes multiple processes (especially for MPI programs). See the basic options section for more details.

3. The least intrusive option is to send the commentary to a network socket. The socket is specified as an IP address and port number pair, like this: `--log-socket=192.168.0.1:12345` if you want to send the output to host IP 192.168.0.1 port 12345 (note: we have no idea if 12345 is a port of pre-existing significance). You can also omit the port number: `--log-socket=192.168.0.1`, in which case a default port of 1500 is used. This default is defined by the constant `VG_CLO_DEFAULT_LOGPORT` in the sources.

   Note, unfortunately, that you have to use an IP address here, rather than a hostname.

   Writing to a network socket is pointless if you don't have something listening at the other end. We provide a simple listener program, `valgrind-listener`, which accepts connections on the specified port and copies whatever it is sent to stdout. Probably someone will tell us this is a horrible security risk. It seems likely that people will write more sophisticated listeners in the fullness of time.

   `valgrind-listener` can accept simultaneous connections from up to 50 Valgrinded processes. In front of each line of output it prints the current number of active connections in round brackets.

   `valgrind-listener` accepts three command-line options:

   `-e --exit-at-zero`

      When the number of connected processes falls back to zero, exit. Without this, it will run forever, that is, until you send it Control-C.

   `--max-connect=INTEGER`

      By default, the listener can connect to up to 50 processes. Occasionally, that number is too small. Use this option to provide a different limit. E.g. `--max-connect=100`.

   `portnumber`

      Changes the port it listens on from the default (1500). The specified port must be in the range 1024 to 65535. The same restriction applies to port numbers specified by a `--log-socket` to Valgrind itself.

   If a Valgrinded process fails to connect to a listener, for whatever reason (the listener isn't running, invalid or unreachable host or port, etc), Valgrind switches back to writing the commentary to stderr. The same goes for any process which loses an established connection to a listener. In other words, killing the listener doesn't kill the processes sending data to it.

Here is an important point about the relationship between the commentary and profiling output from tools. The commentary contains a mix of messages from the Valgrind core and the selected tool. If the tool reports errors, it will report them to the commentary. However, if the tool does profiling, the profile data will be written to a file of some kind, depending on the tool, and independent of what `--log-*` options are in force. The commentary is intended to be a low-bandwidth, human-readable channel. Profiling data, on the other hand, is usually voluminous and not meaningful without further processing, which is why we have chosen this arrangement.

2.4. Reporting of errors

When an error-checking tool detects something bad happening in the program, an error message is written to the commentary. Here's an example from Memcheck:

```
==25832== Invalid read of size 4
==25832==    at 0x8048724: BandMatrix::ReSize(int, int, int) (bogon.cpp:45)
==25832==    by 0x80487AF: main (bogon.cpp:66)
==25832==  Address 0xBFFFF74C is not stack'd, malloc'd or free'd
```

This message says that the program did an illegal 4-byte read of address 0xBFFFF74C, which, as far as Memcheck can tell, is not a valid stack address, nor corresponds to any current heap blocks or recently freed heap blocks. The read is happening at line 45 of `bogon.cpp`, called from line 66 of the same file, etc. For errors associated with an identified (current or freed) heap block, for example reading freed memory, Valgrind reports not only the location where the error happened, but also where the associated heap block was allocated/freed.

Valgrind remembers all error reports. When an error is detected, it is compared against old reports, to see if it is a duplicate. If so, the error is noted, but no further commentary is emitted. This avoids you being swamped with bazillions of duplicate error reports.

If you want to know how many times each error occurred, run with the `-v` option. When execution finishes, all the reports are printed out, along with, and sorted by, their occurrence counts. This makes it easy to see which errors have occurred most frequently.

Errors are reported before the associated operation actually happens. For example, if you're using Memcheck and your program attempts to read from address zero, Memcheck will emit a message to this effect, and your program will then likely die with a segmentation fault.

In general, you should try and fix errors in the order that they are reported. Not doing so can be confusing. For example, a program which copies uninitialised values to several memory locations, and later uses them, will generate several error messages, when run on Memcheck. The first such error message may well give the most direct clue to the root cause of the problem.

The process of detecting duplicate errors is quite an expensive one and can become a significant performance overhead if your program generates huge quantities of errors. To avoid serious problems, Valgrind will simply stop collecting errors after 1,000 different errors have been seen, or 10,000,000 errors in total have been seen. In this situation you might as well stop your program and fix it, because Valgrind won't tell you anything else useful after this. Note that the 1,000/10,000,000 limits apply after suppressed errors are removed. These limits are defined in `m_errormgr.c` and can be increased if necessary.

To avoid this cutoff you can use the `--error-limit=no` option. Then Valgrind will always show errors, regardless of how many there are. Use this option carefully, since it may have a bad effect on performance.

2.5. Suppressing errors

The error-checking tools detect numerous problems in the system libraries, such as the C library, which come pre-installed with your OS. You can't easily fix these, but you don't want to see these errors (and yes, there are many!) So Valgrind reads a list of errors to suppress at startup. A default suppression file is created by the `./configure` script when the system is built.

You can modify and add to the suppressions file at your leisure, or, better, write your own. Multiple suppression files are allowed. This is useful if part of your project contains errors you can't or don't want to fix, yet you don't want to continuously be reminded of them.

Note: By far the easiest way to add suppressions is to use the `--gen-suppressions=yes` option described in Core Command-line Options. This generates suppressions automatically. For best results, though, you may want to edit the output of `--gen-suppressions=yes` by hand, in which case it would be advisable to read through this section.

Each error to be suppressed is described very specifically, to minimise the possibility that a suppression-directive inadvertently suppresses a bunch of similar errors which you did want to see. The suppression mechanism is designed to allow precise yet flexible specification of errors to suppress.

If you use the `-v` option, at the end of execution, Valgrind prints out one line for each used suppression, giving the number of times it got used, its name and the filename and line number where the suppression is defined. Depending on the suppression kind, the filename and line number are optionally followed by additional information (such as the number of blocks and bytes suppressed by a Memcheck leak suppression). Here's the suppressions used by a run of `valgrind -v --tool=memcheck ls -l`:

```
--1610-- used_suppression:      2 dl-hack3-cond-1 /usr/lib/valgrind/default.supp:1234
--1610-- used_suppression:      2 glibc-2.5.x-on-SUSE-10.2-(PPC)-2a /usr/lib/valgrind/defa
```

Multiple suppressions files are allowed. Valgrind loads suppression patterns from `$PREFIX/lib/valgrind/default.supp` unless `--default-suppressions=no` has been specified. You can ask to add suppressions from additional files by specifying `--suppressions=/path/to/file.supp` one or more times.

If you want to understand more about suppressions, look at an existing suppressions file whilst reading the following documentation. The file `glibc-2.3.supp`, in the source distribution, provides some good examples.

Blank and comment lines in a suppression file are ignored. Comment lines are made of 0 or more blanks followed by a # character followed by some text.

Each suppression has the following components:

- First line: its name. This merely gives a handy name to the suppression, by which it is referred to in the summary of used suppressions printed out when a program finishes. It's not important what the name is; any identifying string will do.
- Second line: name of the tool(s) that the suppression is for (if more than one, comma-separated), and the name of the suppression itself, separated by a colon (n.b.: no spaces are allowed), eg:

  ```
  tool_name1,tool_name2:suppression_name
  ```

  Recall that Valgrind is a modular system, in which different instrumentation tools can observe your program whilst it is running. Since different tools detect different kinds of errors, it is necessary to say which tool(s) the suppression is meaningful to.

  Tools will complain, at startup, if a tool does not understand any suppression directed to it. Tools ignore suppressions which are not directed to them. As a result, it is quite practical to put suppressions for all tools into the same suppression file.

- Next line: a small number of suppression types have extra information after the second line (eg. the `Param` suppression for Memcheck)

- Remaining lines: This is the calling context for the error -- the chain of function calls that led to it. There can be up to 24 of these lines.

  Locations may be names of either shared objects, functions, or source lines. They begin with `obj:`, `fun:`, or `src:` respectively. Function, object, and file names to match against may use the wildcard characters `*` and `?`. Source lines are specified using the form `filename[:lineNumber]`.

  Important note: C++ function names must be **mangled**. If you are writing suppressions by hand, use the `--demangle=no` option to get the mangled names in your error messages. An example of a mangled C++ name is `_ZN9QListView4showEv`. This is the form that the GNU C++ compiler uses internally, and the form that must be used in suppression files. The equivalent demangled name, `QListView::show()`, is what you see at the C ++ source code level.

  A location line may also be simply "`...`" (three dots). This is a frame-level wildcard, which matches zero or more frames. Frame level wildcards are useful because they make it easy to ignore varying numbers of uninteresting frames in between frames of interest. That is often important when writing suppressions which are intended to be robust against variations in the amount of function inlining done by compilers.

- Finally, the entire suppression must be between curly braces. Each brace must be the first character on its own line.

A suppression only suppresses an error when the error matches all the details in the suppression. Here's an example:

```
{
  __gconv_transform_ascii_internal/__mbrtowc/mbtowc
  Memcheck:Value4
  fun:__gconv_transform_ascii_internal
  fun:__mbr*toc
  fun:mbtowc
}
```

What it means is: for Memcheck only, suppress a use-of-uninitialised-value error, when the data size is 4, when it occurs in the function `__gconv_transform_ascii_internal`, when that is called from any function of name matching `__mbr*toc`, when that is called from `mbtowc`. It doesn't apply under any other circumstances. The string by which this suppression is identified to the user is `__gconv_transform_ascii_internal/__mbrtowc/mbtowc`.

(See Writing suppression files for more details on the specifics of Memcheck's suppression kinds.)

Another example, again for the Memcheck tool:

```
{
  libX11.so.6.2/libX11.so.6.2/libXaw.so.7.0
  Memcheck:Value4
  obj:/usr/X11R6/lib/libX11.so.6.2
  obj:/usr/X11R6/lib/libX11.so.6.2
  obj:/usr/X11R6/lib/libXaw.so.7.0
}
```

This suppresses any size 4 uninitialised-value error which occurs anywhere in `libX11.so.6.2`, when called from anywhere in the same library, when called from anywhere in `libXaw.so.7.0`. The inexact specification of locations is regrettable, but is about all you can hope for, given that the X11 libraries shipped on the Linux distro on which this example was made have had their symbol tables removed.

An example of the src: specification, again for the Memcheck tool:

```
{
  libX11.so.6.2/libX11.so.6.2/libXaw.so.7.0
  Memcheck:Value4
  src:valid.c:321
}
```

This suppresses any size-4 uninitialised-value error which occurs at line 321 in `valid.c`.

Although the above two examples do not make this clear, you can freely mix `obj:`, `fun:`, and `src:` lines in a suppression.

Finally, here's an example using three frame-level wildcards:

```
{
   a-contrived-example
   Memcheck:Leak
   fun:malloc
   ...
   fun:ddd
   ...
   fun:ccc
   ...
   fun:main
}
```

This suppresses Memcheck memory-leak errors, in the case where the allocation was done by `main` calling (though any number of intermediaries, including zero) `ccc`, calling onwards via `ddd` and eventually to `malloc.`.

2.6. Debuginfod

Valgrind supports the downloading of debuginfo files via debuginfod, an HTTP server for distributing ELF/DWARF debugging information. When a debuginfo file cannot be found locally, Valgrind is able to query debuginfod servers for the file using the file's build-id.

In order to use this feature `debuginfod-find` must be installed and the `$DEBUGINFOD_URLS` environment variable must contain space-separated URLs of debuginfod servers. Valgrind does not support `debuginfod-find` verbose output that is normally enabled with `$DEBUGINFOD_PROGRESS` and `$DEBUGINFOD_VERBOSE`. These environment variables will be ignored. This feature is supported on Linux only.

For more information regarding debuginfod, see Elfutils Debuginfod .

2.7. Core Command-line Options

As mentioned above, Valgrind's core accepts a common set of options. The tools also accept tool-specific options, which are documented separately for each tool.

Valgrind's default settings succeed in giving reasonable behaviour in most cases. We group the available options by rough categories.

2.7.1. Tool-selection Option

The single most important option.

--tool=<toolname> [default: memcheck]

Run the Valgrind tool called toolname, e.g. memcheck, cachegrind, callgrind, helgrind, drd, massif, dhat, lackey, none, exp-bbv, etc.

2.7.2. Basic Options

These options work with all tools.

-h --help

Show help for all options, both for the core and for the selected tool. If the option is repeated it is equivalent to giving --help-debug.

--help-debug

Same as --help, but also lists debugging options which usually are only of use to Valgrind's developers.

--version

Show the version number of the Valgrind core. Tools can have their own version numbers. There is a scheme in place to ensure that tools only execute when the core version is one they are known to work with. This was done to minimise the chances of strange problems arising from tool-vs-core version incompatibilities.

-q, --quiet

Run silently, and only print error messages. Useful if you are running regression tests or have some other automated test machinery.

-v, --verbose

Be more verbose. Gives extra information on various aspects of your program, such as: the shared objects loaded, the suppressions used, the progress of the instrumentation and execution engines, and warnings about unusual behaviour. Repeating the option increases the verbosity level.

--trace-children=<yes|no> [default: no]

When enabled, Valgrind will trace into sub-processes initiated via the exec system call. This is necessary for multi-process programs.

Note that Valgrind does trace into the child of a fork (it would be difficult not to, since fork makes an identical copy of a process), so this option is arguably badly named. However, most children of fork calls immediately call exec anyway.

--trace-children-skip=patt1,patt2,...

This option only has an effect when --trace-children=yes is specified. It allows for some children to be skipped. The option takes a comma separated list of patterns for the names of child executables that Valgrind should not trace into. Patterns may include the metacharacters ? and *, which have the usual meaning.

This can be useful for pruning uninteresting branches from a tree of processes being run on Valgrind. But you should be careful when using it. When Valgrind skips tracing into an executable, it doesn't just skip tracing that executable, it also skips tracing any of that executable's child processes. In other words, the flag doesn't merely cause tracing to stop at the specified executables -- it skips tracing of entire process subtrees rooted at any of the specified executables.

`--trace-children-skip-by-arg=patt1,patt2,...`

This is the same as `--trace-children-skip`, with one difference: the decision as to whether to trace into a child process is made by examining the arguments to the child process, rather than the name of its executable.

`--child-silent-after-fork=<yes|no> [default: no]`

When enabled, Valgrind will not show any debugging or logging output for the child process resulting from a `fork` call. This can make the output less confusing (although more misleading) when dealing with processes that create children. It is particularly useful in conjunction with `--trace-children=`. Use of this option is also strongly recommended if you are requesting XML output (`--xml=yes`), since otherwise the XML from child and parent may become mixed up, which usually makes it useless.

`--vgdb=<no|yes|full> [default: yes]`

Valgrind will provide "gdbserver" functionality when `--vgdb=yes` or `--vgdb=full` is specified. This allows an external GNU GDB debugger to control and debug your program when it runs on Valgrind. `--vgdb=full` incurs significant performance overheads, but provides more precise breakpoints and watchpoints. See Debugging your program using Valgrind's gdbserver and GDB for a detailed description.

If the embedded gdbserver is enabled but no gdb is currently being used, the vgdb command line utility can send "monitor commands" to Valgrind from a shell. The Valgrind core provides a set of Valgrind monitor commands. A tool can optionally provide tool specific monitor commands, which are documented in the tool specific chapter.

`--vgdb-error=<number> [default: 999999999]`

Use this option when the Valgrind gdbserver is enabled with `--vgdb=yes` or `--vgdb=full`. Tools that report errors will wait for "`number`" errors to be reported before freezing the program and waiting for you to connect with GDB. It follows that a value of zero will cause the gdbserver to be started before your program is executed. This is typically used to insert GDB breakpoints before execution, and also works with tools that do not report errors, such as Massif.

`--vgdb-stop-at=<set> [default: none]`

Use this option when the Valgrind gdbserver is enabled with `--vgdb=yes` or `--vgdb=full`. The Valgrind gdbserver will be invoked for each error after `--vgdb-error` have been reported. You can additionally ask the Valgrind gdbserver to be invoked for other events, specified in one of the following ways:

- a comma separated list of one or more of `startup exit abexit valgrindabexit`.

  The values `startup exit valgrindabexit` respectively indicate to invoke gdbserver before your program is executed, after the last instruction of your program, on Valgrind abnormal exit (e.g. internal error, out of memory, ...).

  The option `abexit` is similar to `exit` but tells to invoke gdbserver only when your application exits abnormally (i.e. with an exit code different of 0).

  Note: `startup` and `--vgdb-error=0` will both cause Valgrind gdbserver to be invoked before your program is executed. The `--vgdb-error=0` will in addition cause your program to stop on all subsequent errors.

- `all` to specify the complete set. It is equivalent to `--vgdb-stop-at=startup,exit,abexit,valgrindabexit`.

- `none` for the empty set.

`--track-fds=<yes|no|all> [default: no]`

When enabled, Valgrind will print out a list of open file descriptors on exit or on request, via the gdbserver monitor command `v.info open_fds`. Along with each file descriptor is printed a stack backtrace of where the file was opened and any details relating to the file descriptor such as the file name or socket details. Use `all` to include reporting on `stdin`, `stdout` and `stderr`.

`--modify-fds=<no|high> [default: no]`

When enabled, when the program opens a new file descriptor, the highest available file descriptor is returned instead of the lowest one.

`--time-stamp=<yes|no> [default: no]`

When enabled, each message is preceded with an indication of the elapsed wallclock time since startup, expressed as days, hours, minutes, seconds and milliseconds.

`--log-fd=<number> [default: 2, stderr]`

Specifies that Valgrind should send all of its messages to the specified file descriptor. The default, 2, is the standard error channel (stderr). Note that this may interfere with the client's own use of stderr, as Valgrind's output will be interleaved with any output that the client sends to stderr.

`--log-file=<filename>`

Specifies that Valgrind should send all of its messages to the specified file. If the file name is empty, it causes an abort. There are three special format specifiers that can be used in the file name.

`%p` is replaced with the current process ID. This is very useful for program that invoke multiple processes. WARNING: If you use `--trace-children=yes` and your program invokes multiple processes OR your program forks without calling exec afterwards, and you don't use this specifier (or the `%q` specifier below), the Valgrind output from all those processes will go into one file, possibly jumbled up, and possibly incomplete. Note: If the program forks and calls exec afterwards, Valgrind output of the child from the period between fork and exec will be lost. Fortunately this gap is really tiny for most programs; and modern programs use `posix_spawn` anyway.

`%n` is replaced with a file sequence number unique for this process. This is useful for processes that produces several files from the same filename template.

`%q{FOO}` is replaced with the contents of the environment variable `FOO`. If the `{FOO}` part is malformed, it causes an abort. This specifier is rarely needed, but very useful in certain circumstances (eg. when running MPI programs). The idea is that you specify a variable which will be set differently for each process in the job, for example `BPROC_RANK` or whatever is applicable in your MPI setup. If the named environment variable is not set, it causes an abort. Note that in some shells, the `{` and `}` characters may need to be escaped with a backslash.

`%%` is replaced with `%`.

If an `%` is followed by any other character, it causes an abort.

If the file name specifies a relative file name, it is put in the program's initial working directory: this is the current directory when the program started its execution after the fork or after the exec. If it specifies an absolute file name (ie. starts with '/') then it is put there.

`--log-socket=<ip-address:port-number>`

Specifies that Valgrind should send all of its messages to the specified port at the specified IP address. The port may be omitted, in which case port 1500 is used. If a connection cannot be made to the specified socket, Valgrind falls back to writing output to the standard error (stderr). This option is intended to be used in conjunction with the `valgrind-listener` program. For further details, see the commentary in the manual.

`--enable-debuginfod=<no|yes> [default: yes]`

When enabled Valgrind will attempt to download missing debuginfo from debuginfod servers if space-separated server URLs are present in the `$DEBUGINFOD_URLS` environment variable. This option is supported on Linux only.

2.7.3. Error-related Options

These options are used by all tools that can report errors, e.g. Memcheck, but not Cachegrind.

`--xml=<yes|no> [default: no]`

When enabled, the important parts of the output (e.g. tool error messages) will be in XML format rather than plain text. Furthermore, the XML output will be sent to a different output channel than the plain text output. Therefore, you also must use one of `--xml-fd`, `--xml-file` or `--xml-socket` to specify where the XML is to be sent.

Less important messages will still be printed in plain text, but because the XML output and plain text output are sent to different output channels (the destination of the plain text output is still controlled by `--log-fd`, `--log-file` and `--log-socket`) this should not cause problems.

This option is aimed at making life easier for tools that consume Valgrind's output as input, such as GUI front ends. Currently this option works with Memcheck, Helgrind and DRD. The output format is specified in the file `docs/internals/xml-output-protocol4.txt` in the source tree for Valgrind 3.5.0 or later.

The recommended options for a GUI to pass, when requesting XML output, are: `--xml=yes` to enable XML output, `--xml-file` to send the XML output to a (presumably GUI-selected) file, `--log-file` to send the plain text output to a second GUI-selected file, `--child-silent-after-fork=yes`, and `-q` to restrict the plain text output to critical error messages created by Valgrind itself. For example, failure to read a specified suppressions file counts as a critical error message. In this way, for a successful run the text output file will be empty. But if it isn't empty, then it will contain important information which the GUI user should be made aware of.

`--xml-fd=<number> [default: -1, disabled]`

Specifies that Valgrind should send its XML output to the specified file descriptor. It must be used in conjunction with `--xml=yes`.

`--xml-file=<filename>`

Specifies that Valgrind should send its XML output to the specified file. It must be used in conjunction with `--xml=yes`. Any `%p` or `%q` sequences appearing in the filename are expanded in exactly the same way as they are for `--log-file`. See the description of --log-file for details.

`--xml-socket=<ip-address:port-number>`

Specifies that Valgrind should send its XML output the specified port at the specified IP address. It must be used in conjunction with `--xml=yes`. The form of the argument is the same as that used by `--log-socket`. See the description of `--log-socket` for further details.

`--xml-user-comment=<string>`

Embeds an extra user comment string at the start of the XML output. Only works when `--xml=yes` is specified; ignored otherwise.

`--demangle=<yes|no> [default: yes]`

Enable/disable automatic demangling (decoding) of C++, D, Rust, Java, Ada names. Enabled by default. When enabled, Valgrind will attempt to translate encoded names in the listed languages back to something approaching the original. Note that the callgrind tool always disables Ada demangling in order to differentiate overloaded functions and procedures in the callgraph. The demangler handles symbols mangled by g++ versions 2.X, 3.X and 4.X.

An important fact about demangling is that function names mentioned in suppressions files should be in their mangled form. Valgrind does not demangle function names when searching for applicable suppressions, because to do otherwise would make suppression file contents dependent on the state of Valgrind's demangling machinery, and also slow down suppression matching.

`--num-callers=<number> [default: 12]`

Specifies the maximum number of entries shown in stack traces that identify program locations. Note that errors are commoned up using only the top four function locations (the place in the current function, and that of its three immediate callers). So this doesn't affect the total number of errors reported.

The maximum value for this is 500. Note that higher settings will make Valgrind run a bit more slowly and take a bit more memory, but can be useful when working with programs with deeply-nested call chains.

`--unw-stack-scan-thresh=<number> [default: 0]` , `--unw-stack-scan-frames=<number> [default: 5]`

Stack-scanning support is available only on ARM targets.

These flags enable and control stack unwinding by stack scanning. When the normal stack unwinding mechanisms -- usage of Dwarf CFI records, and frame-pointer following -- fail, stack scanning may be able to recover a stack trace.

Note that stack scanning is an imprecise, heuristic mechanism that may give very misleading results, or none at all. It should be used only in emergencies, when normal unwinding fails, and it is important to nevertheless have stack traces.

Stack scanning is a simple technique: the unwinder reads words from the stack, and tries to guess which of them might be return addresses, by checking to see if they point just after ARM or Thumb call instructions. If so, the word is added to the backtrace.

The main danger occurs when a function call returns, leaving its return address exposed, and a new function is called, but the new function does not overwrite the old address. The result of this is that the backtrace may contain entries for functions which have already returned, and so be very confusing.

A second limitation of this implementation is that it will scan only the page (4KB, normally) containing the starting stack pointer. If the stack frames are large, this may result in only a few (or not even any) being present in the trace. Also, if you are unlucky and have an initial stack pointer near the end of its containing page, the scan may miss all interesting frames.

By default stack scanning is disabled. The normal use case is to ask for it when a stack trace would otherwise be very short. So, to enable it, use `--unw-stack-scan-thresh=number`. This requests Valgrind to try using stack scanning to "extend" stack traces which contain fewer than `number` frames.

If stack scanning does take place, it will only generate at most the number of frames specified by `--unw-stack-scan-frames`. Typically, stack scanning generates so many garbage entries that this value is set to a low value (5) by default. In no case will a stack trace larger than the value specified by `--num-callers` be created.

`--error-limit=<yes|no> [default: yes]`

When enabled, Valgrind stops reporting errors after 10,000,000 in total, or 1,000 different ones, have been seen. This is to stop the error tracking machinery from becoming a huge performance overhead in programs with many errors.

`--error-exitcode=<number> [default: 0]`

Specifies an alternative exit code to return if Valgrind reported any errors in the run. When set to the default value (zero), the return value from Valgrind will always be the return value of the process being simulated. When set to a nonzero value, that value is returned instead, if Valgrind detects any errors. This is useful for using Valgrind as part of an automated test suite, since it makes it easy to detect test cases for which Valgrind has reported errors, just by inspecting return codes. When set to a nonzero value and Valgrind detects no error, the return value of Valgrind will be the return value of the program being simulated.

`--exit-on-first-error=<yes|no> [default: no]`

If this option is enabled, Valgrind exits on the first error. A nonzero exit value must be defined using `--error-exitcode` option. Useful if you are running regression tests or have some other automated test machinery.

`--error-markers=<begin>,<end> [default: none]`

When errors are output as plain text (i.e. XML not used), `--error-markers` instructs to output a line containing the `begin` (`end`) string before (after) each error.

Such marker lines facilitate searching for errors and/or extracting errors in an output file that contain valgrind errors mixed with the program output.

Note that empty markers are accepted. So, only using a begin (or an end) marker is possible.

`--show-error-list=no|yes|all [default: no]`

If this option is yes, for tools that report errors, valgrind will show the list of detected errors and the list of used suppressions at exit. The value all indicates to also show the list of suppressed errors.

Note that at verbosity 2 and above, valgrind automatically shows the list of detected errors and the list of used suppressions at exit, unless `--show-error-list=no` is selected.

`-s`

Specifying `-s` is equivalent to `--show-error-list=yes`.

`--sigill-diagnostics=<yes|no> [default: yes]`

Enable/disable printing of illegal instruction diagnostics. Enabled by default, but defaults to disabled when `--quiet` is given. The default can always be explicitly overridden by giving this option.

When enabled, a warning message will be printed, along with some diagnostics, whenever an instruction is encountered that Valgrind cannot decode or translate, before the program is given a SIGILL signal. Often an illegal instruction indicates a bug in the program or missing support for the particular instruction in Valgrind. But some programs do deliberately try to execute an instruction that might be missing and trap the SIGILL signal to detect processor features. Using this flag makes it possible to avoid the diagnostic output that you would otherwise get in such cases.

`--keep-debuginfo=<yes|no> [default: no]`

When enabled, keep ("archive") symbols and all other debuginfo for unloaded code. This allows saved stack traces to include file/line info for code that has been dlclose'd (or similar). Be careful with this, since it can lead to unbounded memory use for programs which repeatedly load and unload shared objects.

Some tools and some functionalities have only limited support for archived debug info. Memcheck fully supports it. Generally, tools that report errors can use archived debug info to show the error stack traces. The known limitations are: Helgrind's past access stack trace of a race condition is does not use archived debug info. Massif (and more generally the xtree Massif output format) does not make use of archived debug info. Only Memcheck has been (somewhat) tested with `--keep-debuginfo=yes`, so other tools may have unknown limitations.

`--show-below-main=<yes|no> [default: no]`

By default, stack traces for errors do not show any functions that appear beneath `main` because most of the time it's uninteresting C library stuff and/or gobbledygook. Alternatively, if `main` is not present in the stack trace, stack traces will not show any functions below `main`-like functions such as glibc's `__libc_start_main`. Furthermore, if `main`-like functions are present in the trace, they are normalised as `(below main)`, in order to make the output more deterministic.

If this option is enabled, all stack trace entries will be shown and `main`-like functions will not be normalised.

`--fullpath-after=<string> [default: don't show source paths]`

By default Valgrind only shows the filenames in stack traces, but not full paths to source files. When using Valgrind in large projects where the sources reside in multiple different directories, this can be inconvenient. `--fullpath-after` provides a flexible solution to this problem. When this option is present, the path to each source file is shown, with the following all-important caveat: if `string` is found in the path, then the path up to and including `string` is omitted, else the path is shown unmodified. Note that `string` is not required to be a prefix of the path.

For example, consider a file named `/home/janedoe/blah/src/foo/bar/xyzzy.c`. Specifying `--fullpath-after=/home/janedoe/blah/src/` will cause Valgrind to show the name as `foo/bar/xyzzy.c`.

Because the string is not required to be a prefix, `--fullpath-after=src/` will produce the same output. This is useful when the path contains arbitrary machine-generated characters. For example, the path `/my/build/dir/C32A1B47/blah/src/foo/xyzzy` can be pruned to `foo/xyzzy` using `--fullpath-after=/blah/src/`.

If you simply want to see the full path, just specify an empty string: `--fullpath-after=`. This isn't a special case, merely a logical consequence of the above rules.

Finally, you can use `--fullpath-after` multiple times. Any appearance of it causes Valgrind to switch to producing full paths and applying the above filtering rule. Each produced path is compared against all the `--fullpath-after`-specified strings, in the order specified. The first string to match causes the path to be truncated as described above. If none match, the full path is shown. This facilitates chopping off prefixes when the sources are drawn from a number of unrelated directories.

`--extra-debuginfo-path=<path> [default: undefined and unused]`

By default Valgrind searches in several well-known paths for debug objects, such as `/usr/lib/debug/`.

However, there may be scenarios where you may wish to put debug objects at an arbitrary location, such as external storage when running Valgrind on a mobile device with limited local storage. Another example might be a situation where you do not have permission to install debug object packages on the system where you are running Valgrind.

In these scenarios, you may provide an absolute path as an extra, final place for Valgrind to search for debug objects by specifying `--extra-debuginfo-path=/path/to/debug/objects`. The given path will be prepended to the absolute path name of the searched-for object. For example, if Valgrind is looking for the debuginfo for /w/x/y/zz.so and --extra-debuginfo-path=/a/b/c is specified, it will look for a debug object at /a/b/c/w/x/y/zz.so.

This flag should only be specified once. If it is specified multiple times, only the last instance is honoured.

--debuginfo-server=ipaddr:port [default: undefined and unused]

This is a new, experimental, feature introduced in version 3.9.0.

In some scenarios it may be convenient to read debuginfo from objects stored on a different machine. With this flag, Valgrind will query a debuginfo server running on ipaddr and listening on port port, if it cannot find the debuginfo object in the local filesystem.

The debuginfo server must accept TCP connections on port port. The debuginfo server is contained in the source file auxprogs/valgrind-di-server.c. It will only serve from the directory it is started in. port defaults to 1500 in both client and server if not specified.

If Valgrind looks for the debuginfo for /w/x/y/zz.so by using the debuginfo server, it will strip the pathname components and merely request zz.so on the server. That in turn will look only in its current working directory for a matching debuginfo object.

The debuginfo data is transmitted in small fragments (8 KB) as requested by Valgrind. Each block is compressed using LZO to reduce transmission time. The implementation has been tuned for best performance over a single-stage 802.11g (WiFi) network link.

Note that checks for matching primary vs debug objects, using GNU debuglink CRC scheme, are performed even when using the debuginfo server. To disable such checking, you need to also specify --allow-mismatched-debuginfo=yes.

By default the Valgrind build system will build valgrind-di-server for the target platform, which is almost certainly not what you want. So far we have been unable to find out how to get automake/autoconf to build it for the build platform. If you want to use it, you will have to recompile it by hand using the command shown at the top of auxprogs/valgrind-di-server.c.

Valgrind can also download debuginfo via debuginfod. See the DEBUGINFOD section for more information.

--allow-mismatched-debuginfo=no|yes [no]

When reading debuginfo from separate debuginfo objects, Valgrind will by default check that the main and debuginfo objects match, using the GNU debuglink mechanism. This guarantees that it does not read debuginfo from out of date debuginfo objects, and also ensures that Valgrind can't crash as a result of mismatches.

This check can be overridden using --allow-mismatched-debuginfo=yes. This may be useful when the debuginfo and main objects have not been split in the proper way. Be careful when using this, though: it disables all consistency checking, and Valgrind has been observed to crash when the main and debuginfo objects don't match.

--suppressions=<filename> [default: $PREFIX/lib/valgrind/default.supp]

Specifies an extra file from which to read descriptions of errors to suppress. You may use up to 100 extra suppression files.

--gen-suppressions=<yes|no|all> [default: no]

When set to yes, Valgrind will pause after every error shown and print the line:

```
---- Print suppression ? --- [Return/N/n/Y/y/C/c] ----
```

Pressing `Ret`, or `N Ret` or `n Ret`, causes Valgrind continue execution without printing a suppression for this error.

Pressing `Y Ret` or `y Ret` causes Valgrind to write a suppression for this error. You can then cut and paste it into a suppression file if you don't want to hear about the error in the future.

When set to `all`, Valgrind will print a suppression for every reported error, without querying the user.

This option is particularly useful with C++ programs, as it prints out the suppressions with mangled names, as required.

Note that the suppressions printed are as specific as possible. You may want to common up similar ones, by adding wildcards to function names, and by using frame-level wildcards. The wildcarding facilities are powerful yet flexible, and with a bit of careful editing, you may be able to suppress a whole family of related errors with only a few suppressions.

Sometimes two different errors are suppressed by the same suppression, in which case Valgrind will output the suppression more than once, but you only need to have one copy in your suppression file (but having more than one won't cause problems). Also, the suppression name is given as `<insert a suppression name here>`; the name doesn't really matter, it's only used with the `-v` option which prints out all used suppression records.

`--input-fd=<number> [default: 0, stdin]`

When using `--gen-suppressions=yes`, Valgrind will stop so as to read keyboard input from you when each error occurs. By default it reads from the standard input (stdin), which is problematic for programs which close stdin. This option allows you to specify an alternative file descriptor from which to read input.

`--dsymutil=no|yes [yes]`

This option is only relevant when running Valgrind on macOS.

macOS uses a deferred debug information (debuginfo) linking scheme. When object files containing debuginfo are linked into a `.dylib` or an executable, the debuginfo is not copied into the final file. Instead, the debuginfo must be linked manually by running `dsymutil`, a system-provided utility, on the executable or `.dylib`. The resulting combined debuginfo is placed in a directory alongside the executable or `.dylib`, but with the extension `.dSYM`.

With `--dsymutil=no`, Valgrind will detect cases where the `.dSYM` directory is either missing, or is present but does not appear to match the associated executable or `.dylib`, most likely because it is out of date. In these cases, Valgrind will print a warning message but take no further action.

With `--dsymutil=yes`, Valgrind will, in such cases, automatically run `dsymutil` as necessary to bring the debuginfo up to date. For all practical purposes, if you always use `--dsymutil=yes`, then there is never any need to run `dsymutil` manually or as part of your applications's build system, since Valgrind will run it as necessary.

Valgrind will not attempt to run `dsymutil` on any executable or library in `/usr/`, `/bin/`, `/sbin/`, `/opt/`, `/sw/`, `/System/`, `/Library/` or `/Applications/` since `dsymutil` will always fail in such situations. It fails both because the debuginfo for such pre-installed system components is not available anywhere, and also because it would require write privileges in those directories.

Be careful when using `--dsymutil=yes`, since it will cause pre-existing `.dSYM` directories to be silently deleted and re-created. Also note that `dsymutil` is quite slow, sometimes excessively so.

`--max-stackframe=<number> [default: 2000000]`

The maximum size of a stack frame. If the stack pointer moves by more than this amount then Valgrind will assume that the program is switching to a different stack.

You may need to use this option if your program has large stack-allocated arrays. Valgrind keeps track of your program's stack pointer. If it changes by more than the threshold amount, Valgrind assumes your program is switching to a different stack, and Memcheck behaves differently than it would for a stack pointer change smaller than the threshold. Usually this heuristic works well. However, if your program allocates large structures on the stack, this heuristic will be fooled, and Memcheck will subsequently report large numbers of invalid stack accesses. This option allows you to change the threshold to a different value.

You should only consider use of this option if Valgrind's debug output directs you to do so. In that case it will tell you the new threshold you should specify.

In general, allocating large structures on the stack is a bad idea, because you can easily run out of stack space, especially on systems with limited memory or which expect to support large numbers of threads each with a small stack, and also because the error checking performed by Memcheck is more effective for heap-allocated data than for stack-allocated data. If you have to use this option, you may wish to consider rewriting your code to allocate on the heap rather than on the stack.

`--main-stacksize=<number> [default: use current 'ulimit' value]`

Specifies the size of the main thread's stack.

To simplify its memory management, Valgrind reserves all required space for the main thread's stack at startup. That means it needs to know the required stack size at startup.

By default, Valgrind uses the current "ulimit" value for the stack size, or 16 MB, whichever is lower. In many cases this gives a stack size in the range 8 to 16 MB, which almost never overflows for most applications.

If you need a larger total stack size, use `--main-stacksize` to specify it. Only set it as high as you need, since reserving far more space than you need (that is, hundreds of megabytes more than you need) constrains Valgrind's memory allocators and may reduce the total amount of memory that Valgrind can use. This is only really of significance on 32-bit machines.

On Linux, you may request a stack of size up to 2GB. Valgrind will stop with a diagnostic message if the stack cannot be allocated.

`--main-stacksize` only affects the stack size for the program's initial thread. It has no bearing on the size of thread stacks, as Valgrind does not allocate those.

You may need to use both `--main-stacksize` and `--max-stackframe` together. It is important to understand that `--main-stacksize` sets the maximum total stack size, whilst `--max-stackframe` specifies the largest size of any one stack frame. You will have to work out the `--main-stacksize` value for yourself (usually, if your applications segfaults). But Valgrind will tell you the needed `--max-stackframe` size, if necessary.

As discussed further in the description of `--max-stackframe`, a requirement for a large stack is a sign of potential portability problems. You are best advised to place all large data in heap-allocated memory.

`--max-threads=<number> [default: 500]`

By default, Valgrind can handle to up to 500 threads. Occasionally, that number is too small. Use this option to provide a different limit. E.g. `--max-threads=3000`.

`--realloc-zero-bytes-frees=yes|no [default: yes for glibc no otherwise]`

The behaviour of `realloc()` with a size of zero is implementation defined in C17 and undefined in C23. Valgrind tries to work in the same way as the underlying system and C runtime library that it was configured and built on. However, if you use a different C runtime library then this default may be wrong. If the value is `yes` then `realloc` will deallocate the memory and return NULL. If the value is `no` then `realloc` will not deallocate the memory and the size will be handled as though it were one byte.

As an example, if you use Valgrind installed via a package on a Linux distro using GNU libc but link your test executable with musl libc or the JEMalloc library then consider using `--realloc-zero-bytes-frees=no`.

Address Sanitizer has a similar and even wordier option `allocator_frees_and_returns_null_on_realloc_zero`.

2.7.4. malloc-related Options

For tools that use their own version of `malloc` (e.g. Memcheck, Massif, Helgrind, DRD), the following options apply.

`--alignment=<number> [default: 8 or 16, depending on the platform]`

By default Valgrind's `malloc`, `realloc`, etc, return a block whose starting address is 8-byte aligned or 16-byte aligned (the value depends on the platform and matches the platform default). This option allows you to specify a different alignment. The supplied value must be greater than or equal to the default, less than or equal to 4096, and must be a power of two.

`--redzone-size=<number> [default: depends on the tool]`

Valgrind's `malloc, realloc,` etc, add padding blocks before and after each heap block allocated by the program being run. Such padding blocks are called redzones. The default value for the redzone size depends on the tool. For example, Memcheck adds and protects a minimum of 16 bytes before and after each block allocated by the client. This allows it to detect block underruns or overruns of up to 16 bytes.

Increasing the redzone size makes it possible to detect overruns of larger distances, but increases the amount of memory used by Valgrind. Decreasing the redzone size will reduce the memory needed by Valgrind but also reduces the chances of detecting over/underruns, so is not recommended.

`--xtree-memory=none|allocs|full [none]`

Tools replacing Valgrind's `malloc, realloc,` etc, can optionally produce an execution tree detailing which piece of code is responsible for heap memory usage. See Execution Trees for a detailed explanation about execution trees.

When set to `none`, no memory execution tree is produced.

When set to `allocs`, the memory execution tree gives the current number of allocated bytes and the current number of allocated blocks.

When set to `full`, the memory execution tree gives 6 different measurements : the current number of allocated bytes and blocks (same values as for `allocs`), the total number of allocated bytes and blocks, the total number of freed bytes and blocks.

Note that the overhead in cpu and memory to produce an xtree depends on the tool. The overhead in cpu is small for the value `allocs`, as the information needed to produce this report is maintained in any case by the tool. For massif and helgrind, specifying `full` implies to capture a stack trace for each free operation, while normally these tools only capture an allocation stack trace. For Memcheck, the cpu overhead for the value `full` is small, as this can only be used in combination with `--keep-stacktraces=alloc-and-free` or `--keep-stacktraces=alloc-then-free`, which already records a stack trace for each free operation. The memory overhead varies between 5 and 10 words per unique stacktrace in the xtree, plus the memory needed to record the stack trace for the free operations, if needed specifically for the xtree.

`--xtree-memory-file=<filename> [default: xtmemory.kcg.%p]`

Specifies that Valgrind should produce the xtree memory report in the specified file. Any `%p` or `%q` sequences appearing in the filename are expanded in exactly the same way as they are for `--log-file`. See the description of --log-file for details.

If the filename contains the extension `.ms`, then the produced file format will be a massif output file format. If the filename contains the extension `.kcg` or no extension is provided or recognised, then the produced file format will be a callgrind output format.

See Execution Trees for a detailed explanation about execution trees formats.

2.7.5. Uncommon Options

These options apply to all tools, as they affect certain obscure workings of the Valgrind core. Most people won't need to use them.

`--smc-check=<none|stack|all|all-non-file> [default: all-non-file for x86/amd64/s390x, stack for other archs]`

This option controls Valgrind's detection of self-modifying code. If no checking is done, when a program executes some code, then overwrites it with new code, and executes the new code, Valgrind will continue to execute the translations it made for the old code. This will likely lead to incorrect behaviour and/or crashes.

For "modern" architectures -- anything that's not x86, amd64 or s390x -- the default is `stack`. This is because a correct program must take explicit action to reestablish D-I cache coherence following code modification. Valgrind observes and honours such actions, with the result that self-modifying code is transparently handled with zero extra cost.

For x86, amd64 and s390x, the program is not required to notify the hardware of required D-I coherence syncing. Hence the default is `all-non-file`, which covers the normal case of generating code into an anonymous (non-file-backed) mmap'd area.

The meanings of the four available settings are as follows. No detection (`none`), detect self-modifying code on the stack (which is used by GCC to implement nested functions) (`stack`), detect self-modifying code everywhere (`all`), and detect self-modifying code everywhere except in file-backed mappings (`all-non-file`).

Running with `all` will slow Valgrind down noticeably. Running with `none` will rarely speed things up, since very little code gets dynamically generated in most programs. The `VALGRIND_DISCARD_TRANSLATIONS` client request is an alternative to `--smc-check=all` and `--smc-check=all-non-file` that requires more programmer effort but allows Valgrind to run your program faster, by telling it precisely when translations need to be re-made.

`--smc-check=all-non-file` provides a cheaper but more limited version of `--smc-check=all`. It adds checks to any translations that do not originate from file-backed memory mappings. Typical applications that generate code, for example JITs in web browsers, generate code into anonymous mmaped areas, whereas the "fixed" code of the browser always lives in file-backed mappings. `--smc-check=all-non-file` takes advantage of this observation, limiting the overhead of checking to code which is likely to be JIT generated.

`--read-inline-info=<yes|no> [default: see below]`

When enabled, Valgrind will read information about inlined function calls from DWARF3 debug info. This slows Valgrind startup and makes it use more memory (typically for each inlined piece of code, 6 words and space for the function name), but it results in more descriptive stacktraces. Currently, this functionality is enabled by default only for Linux, FreeBSD, Android and Solaris targets and only for the tools Memcheck, Massif, Helgrind and DRD. Here is an example of some stacktraces with `--read-inline-info=no`:

```
==15380== Conditional jump or move depends on uninitialised value(s)
==15380==    at 0x80484EA: main (inlinfo.c:6)
==15380==
==15380== Conditional jump or move depends on uninitialised value(s)
==15380==    at 0x8048550: fun_noninline (inlinfo.c:6)
==15380==    by 0x804850E: main (inlinfo.c:34)
==15380==
==15380== Conditional jump or move depends on uninitialised value(s)
==15380==    at 0x8048520: main (inlinfo.c:6)
```

And here are the same errors with `--read-inline-info=yes`:

```
==15377== Conditional jump or move depends on uninitialised value(s)
==15377==    at 0x80484EA: fun_d (inlinfo.c:6)
==15377==    by 0x80484EA: fun_c (inlinfo.c:14)
==15377==    by 0x80484EA: fun_b (inlinfo.c:20)
==15377==    by 0x80484EA: fun_a (inlinfo.c:26)
==15377==    by 0x80484EA: main (inlinfo.c:33)
==15377==
==15377== Conditional jump or move depends on uninitialised value(s)
==15377==    at 0x8048550: fun_d (inlinfo.c:6)
==15377==    by 0x8048550: fun_noninline (inlinfo.c:41)
==15377==    by 0x804850E: main (inlinfo.c:34)
==15377==
==15377== Conditional jump or move depends on uninitialised value(s)
==15377==    at 0x8048520: fun_d (inlinfo.c:6)
==15377==    by 0x8048520: main (inlinfo.c:35)
```

`--read-var-info=<yes|no> [default: no]`

When enabled, Valgrind will read information about variable types and locations from DWARF3 debug info. This slows Valgrind startup significantly and makes it use significantly more memory, but for the tools that can take advantage of it (Memcheck, Helgrind, DRD) it can result in more precise error messages. For example, here are some standard errors issued by Memcheck:

```
==15363== Uninitialised byte(s) found during client check request
==15363==    at 0x80484A9: croak (varinfo1.c:28)
==15363==    by 0x8048544: main (varinfo1.c:55)
==15363==  Address 0x80497f7 is 7 bytes inside data symbol "global_i2"
==15363==
==15363== Uninitialised byte(s) found during client check request
==15363==    at 0x80484A9: croak (varinfo1.c:28)
==15363==    by 0x8048550: main (varinfo1.c:56)
==15363==  Address 0xbea0d0cc is on thread 1's stack
==15363==  in frame #1, created by main (varinfo1.c:45)
```

And here are the same errors with `--read-var-info=yes`:

```
==15370== Uninitialised byte(s) found during client check request
==15370==    at 0x80484A9: croak (varinfo1.c:28)
==15370==    by 0x8048544: main (varinfo1.c:55)
```

```
==15370==  Location 0x80497f7 is 0 bytes inside global_i2[7],
==15370==  a global variable declared at varinfo1.c:41
==15370==
==15370== Uninitialised byte(s) found during client check request
==15370==    at 0x80484A9: croak (varinfo1.c:28)
==15370==    by 0x8048550: main (varinfo1.c:56)
==15370==  Location 0xbeb4a0cc is 0 bytes inside local var "local"
==15370==  declared at varinfo1.c:46, in frame #1 of thread 1
```

`--vgdb-poll=<number> [default: 5000]`

As part of its main loop, the Valgrind scheduler will poll to check if some activity (such as an external command or some input from a gdb) has to be handled by gdbserver. This activity poll will be done after having run the given number of basic blocks (or slightly more than the given number of basic blocks). This poll is quite cheap so the default value is set relatively low. You might further decrease this value if vgdb cannot use ptrace system call to interrupt Valgrind if all threads are (most of the time) blocked in a system call.

`--vgdb-shadow-registers=no|yes [default: no]`

When activated, gdbserver will expose the Valgrind shadow registers to GDB. With this, the value of the Valgrind shadow registers can be examined or changed using GDB. Exposing shadow registers only works with GDB version 7.1 or later.

`--vgdb-prefix=<prefix> [default: /tmp/vgdb-pipe]`

To communicate with gdb/vgdb, the Valgrind gdbserver creates 3 files (2 named FIFOs and a mmap shared memory file). The prefix option controls the directory and prefix for the creation of these files.

`--run-libc-freeres=<yes|no> [default: yes]`

This option is only relevant when running Valgrind on Linux with GNU libc.

The GNU C library (`libc.so`), which is used by all programs, may allocate memory for its own uses. Usually it doesn't bother to free that memory when the program ends—there would be no point, since the Linux kernel reclaims all process resources when a process exits anyway, so it would just slow things down.

The glibc authors realised that this behaviour causes leak checkers, such as Valgrind, to falsely report leaks in glibc, when a leak check is done at exit. In order to avoid this, they provided a routine called `__libc_freeres` specifically to make glibc release all memory it has allocated. Memcheck therefore tries to run `__libc_freeres` at exit.

Unfortunately, in some very old versions of glibc, `__libc_freeres` is sufficiently buggy to cause segmentation faults. This was particularly noticeable on Red Hat 7.1. So this option is provided in order to inhibit the run of `__libc_freeres`. If your program seems to run fine on Valgrind, but segfaults at exit, you may find that `--run-libc-freeres=no` fixes that, although at the cost of possibly falsely reporting space leaks in `libc.so`.

`--run-cxx-freeres=<yes|no> [default: yes]`

This option is only relevant when running Valgrind on Linux, FreeBSD or Solaris C++ programs using libstdc++.

The GNU Standard C++ library (`libstdc++.so`), which is used by all C++ programs compiled with g++, may allocate memory for its own uses. Usually it doesn't bother to free that memory when the program ends—there would be no point, since the kernel reclaims all process resources when a process exits anyway, so it would just slow things down.

The gcc authors realised that this behaviour causes leak checkers, such as Valgrind, to falsely report leaks in libstdc++, when a leak check is done at exit. In order to avoid this, they provided a routine called `__gnu_cxx::__freeres` specifically to make libstdc++ release all memory it has allocated. Memcheck therefore tries to run `__gnu_cxx::__freeres` at exit.

For the sake of flexibility and unforeseen problems with `__gnu_cxx::__freeres`, option `--run-cxx-freeres=no` exists, although at the cost of possibly falsely reporting space leaks in `libstdc++.so`.

`--sim-hints=hint1,hint2,...`

Pass miscellaneous hints to Valgrind which slightly modify the simulated behaviour in nonstandard or dangerous ways, possibly to help the simulation of strange features. By default no hints are enabled. Use with caution! Currently known hints are:

- `lax-ioctls:` Be very lax about ioctl handling; the only assumption is that the size is correct. Doesn't require the full buffer to be initialised when writing. Without this, using some device drivers with a large number of strange ioctl commands becomes very tiresome.
- `fuse-compatible:` Enable special handling for certain system calls that may block in a FUSE file-system. This may be necessary when running Valgrind on a multi-threaded program that uses one thread to manage a FUSE file-system and another thread to access that file-system.
- `enable-outer:` Enable some special magic needed when the program being run is itself Valgrind.
- `no-inner-prefix:` Disable printing a prefix > in front of each stdout or stderr output line in an inner Valgrind being run by an outer Valgrind. This is useful when running Valgrind regression tests in an outer/ inner setup. Note that the prefix > will always be printed in front of the inner debug logging lines.
- `no-nptl-pthread-stackcache:` This hint is only relevant when running Valgrind on Linux; it is ignored on FreeBSD, Solaris and macOS.

  The GNU glibc pthread library (`libpthread.so`), which is used by pthread programs, maintains a cache of pthread stacks. When a pthread terminates, the memory used for the pthread stack and some thread local storage related data structure are not always directly released. This memory is kept in a cache (up to a certain size), and is re-used if a new thread is started.

  This cache causes the helgrind tool to report some false positive race condition errors on this cached memory, as helgrind does not understand the internal glibc cache synchronisation primitives. So, when using helgrind, disabling the cache helps to avoid false positive race conditions, in particular when using thread local storage variables (e.g. variables using the `__thread` qualifier).

  When using the memcheck tool, disabling the cache ensures the memory used by glibc to handle __thread variables is directly released when a thread terminates.

  Note: Valgrind disables the cache using some internal knowledge of the glibc stack cache implementation and by examining the debug information of the pthread library. This technique is thus somewhat fragile and might not work for all glibc versions. This has been successfully tested with various glibc versions (e.g. 2.11, 2.16, 2.18) on various platforms.
- `lax-doors:` (Solaris only) Be very lax about door syscall handling over unrecognised door file descriptors. Does not require that full buffer is initialised when writing. Without this, programs using libdoor(3LIB) functionality with completely proprietary semantics may report large number of false positives.
- `fallback-llsc:` (MIPS and ARM64 only): Enables an alternative implementation of Load-Linked (LL) and Store-Conditional (SC) instructions. The standard implementation gives more correct behaviour, but can cause indefinite looping on certain processor implementations that are intolerant of extra memory references between LL and SC. So far this is known only to happen on Cavium 3 cores. You should not need to use this

flag, since the relevant cores are detected at startup and the alternative implementation is automatically enabled if necessary. There is no equivalent anti-flag: you cannot force-disable the alternative implementation, if it is automatically enabled. The underlying problem exists because the "standard" implementation of LL and SC is done by copying through LL and SC instructions into the instrumented code. However, tools may insert extra instrumentation memory references in between the LL and SC instructions. These memory references are not present in the original uninstrumented code, and their presence in the instrumented code can cause the SC instructions to persistently fail, leading to indefinite looping in LL-SC blocks. The alternative implementation gives correct behaviour of LL and SC instructions between threads in a process, up to and including the ABA scenario. It also gives correct behaviour between a Valgrinded thread and a non-Valgrinded thread running in a different process, that communicate via shared memory, but only up to and including correct CAS behaviour -- in this case the ABA scenario may not be correctly handled.

`--scheduling-quantum=<number> [default: 100000]`

The `--scheduling-quantum` option controls the maximum number of basic blocks executed by a thread before releasing the lock used by Valgrind to serialise thread execution. Smaller values give finer interleaving but increases the scheduling overhead. Finer interleaving can be useful to reproduce race conditions with helgrind or DRD. For more details about the Valgrind thread serialisation scheme and its impact on performance and thread scheduling, see Scheduling and Multi-Thread Performance.

`--fair-sched=<no|yes|try> [default: no]`

The `--fair-sched` option controls the locking mechanism used by Valgrind to serialise thread execution. The locking mechanism controls the way the threads are scheduled, and different settings give different trade-offs between fairness and performance. For more details about the Valgrind thread serialisation scheme and its impact on performance and thread scheduling, see Scheduling and Multi-Thread Performance.

- The value `--fair-sched=yes` activates a fair scheduler. In short, if multiple threads are ready to run, the threads will be scheduled in a round robin fashion. This mechanism is not available on all platforms or Linux versions. If not available, using `--fair-sched=yes` will cause Valgrind to terminate with an error.

  You may find this setting improves overall responsiveness if you are running an interactive multithreaded program, for example a web browser, on Valgrind.

- The value `--fair-sched=try` activates fair scheduling if available on the platform. Otherwise, it will automatically fall back to `--fair-sched=no`.

- The value `--fair-sched=no` activates a scheduler which does not guarantee fairness between threads ready to run, but which in general gives the highest performance.

`--kernel-variant=variant1,variant2,...`

Handle system calls and ioctls arising from minor variants of the default kernel for this platform. This is useful for running on hacked kernels or with kernel modules which support nonstandard ioctls, for example. Use with caution. If you don't understand what this option does then you almost certainly don't need it. Currently known variants are:

- `bproc`: support the `sys_broc` system call on x86. This is for running on BProc, which is a minor variant of standard Linux which is sometimes used for building clusters.

- `android-no-hw-tls`: some versions of the Android emulator for ARM do not provide a hardware TLS (thread-local state) register, and Valgrind crashes at startup. Use this variant to select software support for TLS.

- `android-gpu-sgx5xx`: use this to support handling of proprietary ioctls for the PowerVR SGX 5XX series of GPUs on Android devices. Failure to select this does not cause stability problems, but may cause Memcheck to report false errors after the program performs GPU-specific ioctls.

- `android-gpu-adreno3xx`: similarly, use this to support handling of proprietary ioctls for the Qualcomm Adreno 3XX series of GPUs on Android devices.

`--merge-recursive-frames=<number> [default: 0]`

Some recursive algorithms, for example balanced binary tree implementations, create many different stack traces, each containing cycles of calls. A cycle is defined as two identical program counter values separated by zero or more other program counter values. Valgrind may then use a lot of memory to store all these stack traces. This is a poor use of memory considering that such stack traces contain repeated uninteresting recursive calls instead of more interesting information such as the function that has initiated the recursive call.

The option `--merge-recursive-frames=<number>` instructs Valgrind to detect and merge recursive call cycles having a size of up to `<number>` frames. When such a cycle is detected, Valgrind records the cycle in the stack trace as a unique program counter.

The value 0 (the default) causes no recursive call merging. A value of 1 will cause stack traces of simple recursive algorithms (for example, a factorial implementation) to be collapsed. A value of 2 will usually be needed to collapse stack traces produced by recursive algorithms such as binary trees, quick sort, etc. Higher values might be needed for more complex recursive algorithms.

Note: recursive calls are detected by analysis of program counter values. They are not detected by looking at function names.

`--num-transtab-sectors=<number> [default: 6 for Android platforms, 16 for all others]`

Valgrind translates and instruments your program's machine code in small fragments (basic blocks). The translations are stored in a translation cache that is divided into a number of sections (sectors). If the cache is full, the sector containing the oldest translations is emptied and reused. If these old translations are needed again, Valgrind must re-translate and re-instrument the corresponding machine code, which is expensive. If the "executed instructions" working set of a program is big, increasing the number of sectors may improve performance by reducing the number of re-translations needed. Sectors are allocated on demand. Once allocated, a sector can never be freed, and occupies considerable space, depending on the tool and the value of `--avg-transtab-entry-size` (about 40 MB per sector for Memcheck). Use the option `--stats=yes` to obtain precise information about the memory used by a sector and the allocation and recycling of sectors.

`--avg-transtab-entry-size=<number> [default: 0, meaning use tool provided default]`

Average size of translated basic block. This average size is used to dimension the size of a sector. Each tool provides a default value to be used. If this default value is too small, the translation sectors will become full too quickly. If this default value is too big, a significant part of the translation sector memory will be unused. Note that the average size of a basic block translation depends on the tool, and might depend on tool options. For example, the memcheck option `--track-origins=yes` increases the size of the basic block translations. Use `--avg-transtab-entry-size` to tune the size of the sectors, either to gain memory or to avoid too many retranslations.

`--aspace-minaddr=<address> [default: depends on the platform]`

To avoid potential conflicts with some system libraries, Valgrind does not use the address space below `--aspace-minaddr` value, keeping it reserved in case a library specifically requests memory in this region. So, some "pessimistic" value is guessed by Valgrind depending on the platform. On linux, by default, Valgrind avoids using the first 64MB even if typically there is no conflict in this complete zone. You can use the option `--aspace-minaddr` to have your memory hungry application benefitting from more of this lower memory. On the other hand, if you encounter a conflict, increasing aspace-minaddr value might solve it. Conflicts will typically manifest themselves with mmap failures in the low range of the address space. The provided `address` must be page aligned and must be equal or bigger to 0x1000 (4KB). To find the default value on your platform, do something such as `valgrind -d -d date 2>&1 | grep -i minaddr`. Values lower than 0x10000 (64KB) are known to create problems on some distributions.

`--valgrind-stacksize=<number> [default: 1MB]`

For each thread, Valgrind needs its own 'private' stack. The default size for these stacks is largely dimensioned, and so should be sufficient in most cases. In case the size is too small, Valgrind will segfault. Before segfaulting, a warning might be produced by Valgrind when approaching the limit.

Use the option `--valgrind-stacksize` if such an (unlikely) warning is produced, or Valgrind dies due to a segmentation violation. Such segmentation violations have been seen when demangling huge C++ symbols.

If your application uses many threads and needs a lot of memory, you can gain some memory by reducing the size of these Valgrind stacks using the option `--valgrind-stacksize`.

`--show-emwarns=<yes|no> [default: no]`

When enabled, Valgrind will emit warnings about its CPU emulation in certain cases. These are usually not interesting.

`--require-text-symbol=:sonamepatt:fnnamepatt`

When a shared object whose soname matches `sonamepatt` is loaded into the process, examine all the text symbols it exports. If none of those match `fnnamepatt`, print an error message and abandon the run. This makes it possible to ensure that the run does not continue unless a given shared object contains a particular function name.

Both `sonamepatt` and `fnnamepatt` can be written using the usual `?` and `*` wildcards. For example: `":*libc.so*:foo?bar"`. You may use characters other than a colon to separate the two patterns. It is only important that the first character and the separator character are the same. For example, the above example could also be written `"Q*libc.so*Qfoo?bar"`. Multiple `--require-text-symbol` flags are allowed, in which case shared objects that are loaded into the process will be checked against all of them.

The purpose of this is to support reliable usage of marked-up libraries. For example, suppose we have a version of GCC's `libgomp.so` which has been marked up with annotations to support Helgrind. It is only too easy and confusing to load the wrong, un-annotated `libgomp.so` into the application. So the idea is: add a text symbol in the marked-up library, for example `annotated_for_helgrind_3_6`, and then give the flag `--require-text-symbol=:*libgomp*so*:annotated_for_helgrind_3_6` so that when `libgomp.so` is loaded, Valgrind scans its symbol table, and if the symbol isn't present the run is aborted, rather than continuing silently with the un-marked-up library. Note that you should put the entire flag in quotes to stop shells expanding up the `*` and `?` wildcards.

`--soname-synonyms=syn1=pattern1,syn2=pattern2,...`

When a shared library is loaded, Valgrind checks for functions in the library that must be replaced or wrapped. For example, Memcheck replaces some string and memory functions (strchr, strlen, strcpy, memchr, memcpy, memmove, etc.) with its own versions. Such replacements are normally done only in shared libraries whose soname matches a predefined soname pattern (e.g. `libc.so*` on linux). By default, no replacement is done for a statically linked binary or for alternative libraries, except for the allocation functions (malloc, free, calloc, memalign, realloc, operator new, operator delete, etc.) Such allocation functions are intercepted by default in any shared library or in the executable if they are exported as global symbols. This means that if a replacement allocation library such as tcmalloc is found, its functions are also intercepted by default. In some cases, the replacements allow `--soname-synonyms` to specify one additional synonym pattern, giving flexibility in the replacement. Or to prevent interception of all public allocation symbols.

Currently, this flexibility is only allowed for the malloc related functions, using the synonym `somalloc`. This synonym is usable for all tools doing standard replacement of malloc related functions (e.g. memcheck, helgrind, drd, massif, dhat).

- Alternate malloc library: to replace the malloc related functions in a specific alternate library with soname `mymalloclib.so` (and not in any others), give the option `--soname-synonyms=somalloc=mymalloclib.so`. A pattern can be used to match multiple libraries sonames. For example, `--soname-synonyms=somalloc=*tcmalloc*` will match the soname of all variants of the tcmalloc library (native, debug, profiled, ... tcmalloc variants).

  Note: the soname of a elf shared library can be retrieved using the readelf utility.

- Replacements in a statically linked library are done by using the `NONE` pattern. For example, if you link with `libtcmalloc.a`, and only want to intercept the malloc related functions in the executable (and standard libraries) themselves, but not any other shared libraries, you can give the option `--soname-synonyms=somalloc=NONE`. Note that a NONE pattern will match the main executable and any shared library having no soname.

- To only intercept allocation symbols in the default system libraries, but not in any other shared library or the executable defining public malloc or operator new related functions use a non-existing library name like `--soname-synonyms=somalloc=nouserintercepts` (where `nouserintercepts` can be any non-existing library name).

- Shared library of the dynamic (runtime) linker is excluded from searching for global public symbols, such as those for the malloc related functions (identified by `somalloc` synonym).

`--progress-interval=<number> [default: 0, meaning 'disabled']`

This is an enhancement to Valgrind's debugging output. It is unlikely to be of interest to end users.

When `number` is set to a non-zero value, Valgrind will print a one-line progress summary every `number` seconds. Valid settings for `number` are between 0 and 3600 inclusive. Here's some example output with `number` set to 10:

```
PROGRESS: U 110s, W 113s, 97.3% CPU, EvC 414.79M, TIn 616.7k, TOut 0.5k, #thr 67
PROGRESS: U 120s, W 124s, 96.8% CPU, EvC 505.27M, TIn 636.6k, TOut 3.0k, #thr 64
PROGRESS: U 130s, W 134s, 97.0% CPU, EvC 574.90M, TIn 657.5k, TOut 3.0k, #thr 63
```

Each line shows:

- `U`: total user time
- `W`: total wallclock time
- `CPU`: overall average cpu use
- `EvC`: number of event checks. An event check is a backwards branch in the simulated program, so this is a measure of forward progress of the program
- `TIn`: number of code blocks instrumented by the JIT
- `TOut`: number of instrumented code blocks that have been thrown away
- `#thr`: number of threads in the program

From the progress of these, it is possible to observe:

- when the program is compute bound (`TIn` rises slowly, `EvC` rises rapidly)

- when the program is in a spinloop (`TIn`/`TOut` fixed, `EvC` rises rapidly)
- when the program is JIT-bound (`TIn` rises rapidly)
- when the program is rapidly discarding code (`TOut` rises rapidly)
- when the program is about to achieve some expected state (`EvC` arrives at some value you expect)
- when the program is idling (`U` rises more slowly than `W`)

2.7.6. Debugging Options

There are also some options for debugging Valgrind itself. You shouldn't need to use them in the normal run of things. If you wish to see the list, use the `--help-debug` option.

If you wish to debug your program rather than debugging Valgrind itself, then you should use the options `--vgdb=yes` or `--vgdb=full`.

2.7.7. Setting Default Options

Note that Valgrind also reads options from three places:

1. The file `~/.valgrindrc`
2. The environment variable `$VALGRIND_OPTS`
3. The file `./.valgrindrc`

These are processed in the given order, before the command-line options. Options processed later override those processed earlier; for example, options in `./.valgrindrc` will take precedence over those in `~/.valgrindrc`.

Please note that the `./.valgrindrc` file is ignored if it is not a regular file, or is marked as world writeable, or is not owned by the current user. This is because the `./.valgrindrc` can contain options that are potentially harmful or can be used by a local attacker to execute code under your user account.

Any tool-specific options put in `$VALGRIND_OPTS` or the `.valgrindrc` files should be prefixed with the tool name and a colon. For example, if you want Memcheck to always do leak checking, you can put the following entry in `~/.valgrindrc`:

```
--memcheck:leak-check=yes
```

This will be ignored if any tool other than Memcheck is run. Without the `memcheck:` part, this will cause problems if you select other tools that don't understand `--leak-check=yes`.

2.7.8. Dynamically Changing Options

The value of some command line options can be changed dynamically while your program is running under Valgrind.

The dynamically changeable options of the valgrind core and a given tool can be listed using option `--help-dyn-options`, for example:

```
$ valgrind --tool=memcheck --help-dyn-options
  dynamically changeable options:
    -v -q -d --stats --vgdb=no --vgdb=yes --vgdb=full --vgdb-poll --vgdb-error
    --vgdb-stop-at --error-markers --show-error-list -s --show-below-main
```

```
    --time-stamp --trace-children --child-silent-after-fork --trace-sched
    --trace-signals --trace-symtab --trace-cfi --debug-dump=syms
    --debug-dump=line --debug-dump=frames --trace-redir --trace-syscalls
    --sym-offsets --progress-interval --merge-recursive-frames
    --vex-iropt-verbosity --suppressions --trace-flags --trace-notbelow
    --trace-notabove --profile-flags --gen-suppressions=no
    --gen-suppressions=yes --gen-suppressions=all --errors-for-leak-kinds
    --show-leak-kinds --leak-check-heuristics --show-reachable
    --show-possibly-lost --freelist-vol --freelist-big-blocks --leak-check=no
    --leak-check=summary --leak-check=yes --leak-check=full --ignore-ranges
    --ignore-range-below-sp --show-mismatched-frees
valgrind: Use --help for more information.
$
```

The dynamic options can be changed the following ways:

1. From the shell, using vgdb and the monitor command `v.clo`:

   ```
   $ vgdb "v.clo --trace-children=yes --child-silent-after-fork=no"
   sending command v.clo --trace-children=yes --child-silent-after-fork=no to pid 4404
   $
   ```

   Note: you must use double quotes around the monitor command to avoid vgdb interpreting the valgrind options as its own options.

2. From gdb, using the monitor command `v.clo`:

   ```
   (gdb) monitor v.clo --trace-children=yes --child-silent-after-fork=no
   (gdb)
   ```

3. From your program, using the client request `VALGRIND_CLO_CHANGE(option)`:

   ```
         VALGRIND_CLO_CHANGE ("--trace-children=yes");
         VALGRIND_CLO_CHANGE ("--child-silent-after-fork=no");
   ```

Dynamically changeable options can be used in various circumstances, such as changing the valgrind behaviour during execution, loading suppression files as part of shared library initialisation, change or set valgrind options in child processes, ...

2.8. Support for Threads

Threaded programs are fully supported.

The main thing to point out with respect to threaded programs is that your program will use the native threading library, but Valgrind serialises execution so that only one (kernel) thread is running at a time. This approach avoids the horrible implementation problems of implementing a truly multithreaded version of Valgrind, but it does mean that threaded apps never use more than one CPU simultaneously, even if you have a multiprocessor or multicore machine.

Valgrind doesn't schedule the threads itself. It merely ensures that only one thread runs at once, using a simple locking scheme. The actual thread scheduling remains under control of the OS kernel. What this does mean, though, is that your program will see very different scheduling when run on Valgrind than it does when running normally. This is both because Valgrind is serialising the threads, and because the code runs so much slower than normal.

This difference in scheduling may cause your program to behave differently, if you have some kind of concurrency, critical race, locking, or similar, bugs. In that case you might consider using the tools Helgrind and/or DRD to track them down.

On Linux, Valgrind also supports direct use of the `clone` system call, `futex` and so on. `clone` is supported where either everything is shared (a thread) or nothing is shared (fork-like); partial sharing will fail.

2.8.1. Scheduling and Multi-Thread Performance

A thread executes code only when it holds the abovementioned lock. After executing some number of instructions, the running thread will release the lock. All threads ready to run will then compete to acquire the lock.

The `--fair-sched` option controls the locking mechanism used to serialise thread execution.

The default pipe based locking mechanism (`--fair-sched=no`) is available on all platforms. Pipe based locking does not guarantee fairness between threads: it is quite likely that a thread that has just released the lock reacquires it immediately, even though other threads are ready to run. When using pipe based locking, different runs of the same multithreaded application might give very different thread scheduling.

An alternative locking mechanism, based on futexes, is available on some platforms. If available, it is activated by `--fair-sched=yes` or `--fair-sched=try`. Futex based locking ensures fairness (round-robin scheduling) between threads: if multiple threads are ready to run, the lock will be given to the thread which first requested the lock. Note that a thread which is blocked in a system call (e.g. in a blocking read system call) has not (yet) requested the lock: such a thread requests the lock only after the system call is finished.

The fairness of the futex based locking produces better reproducibility of thread scheduling for different executions of a multithreaded application. This better reproducibility is particularly helpful when using Helgrind or DRD.

Valgrind's use of thread serialisation implies that only one thread at a time may run. On a multiprocessor/multicore system, the running thread is assigned to one of the CPUs by the OS kernel scheduler. When a thread acquires the lock, sometimes the thread will be assigned to the same CPU as the thread that just released the lock. Sometimes, the thread will be assigned to another CPU. When using pipe based locking, the thread that just acquired the lock will usually be scheduled on the same CPU as the thread that just released the lock. With the futex based mechanism, the thread that just acquired the lock will more often be scheduled on another CPU.

Valgrind's thread serialisation and CPU assignment by the OS kernel scheduler can interact badly with the CPU frequency scaling available on many modern CPUs. To decrease power consumption, the frequency of a CPU or core is automatically decreased if the CPU/core has not been used recently. If the OS kernel often assigns the thread which just acquired the lock to another CPU/core, it is quite likely that this CPU/core is currently at a low frequency. The frequency of this CPU will be increased after some time. However, during this time, the (only) running thread will have run at the low frequency. Once this thread has run for some time, it will release the lock. Another thread will acquire this lock, and might be scheduled again on another CPU whose clock frequency was decreased in the meantime.

The futex based locking causes threads to change CPUs/cores more often. So, if CPU frequency scaling is activated, the futex based locking might decrease significantly the performance of a multithreaded app running under Valgrind. Performance losses of up to 50% degradation have been observed, as compared to running on a machine for which CPU frequency scaling has been disabled. The pipe based locking locking scheme also interacts badly with CPU frequency scaling, with performance losses in the range 10..20% having been observed.

To avoid such performance degradation, you should indicate to the kernel that all CPUs/cores should always run at maximum clock speed. Depending on your Linux distribution, CPU frequency scaling may be controlled using a graphical interface or using command line such as `cpufreq-selector` or `cpufreq-set`.

An alternative way to avoid these problems is to tell the OS scheduler to tie a Valgrind process to a specific (fixed) CPU using the `taskset` command. This should ensure that the selected CPU does not fall below its maximum frequency setting so long as any thread of the program has work to do.

2.9. Handling of Signals

Valgrind has a fairly complete signal implementation. It should be able to cope with any POSIX-compliant use of signals.

If you're using signals in clever ways (for example, catching SIGSEGV, modifying page state and restarting the instruction), you're probably relying on precise exceptions. In this case, you will need to use `--vex-iropt-register-updates=allregs-at-mem-access` or `--vex-iropt-register-updates=allregs-at-each-insn`.

If your program dies as a result of a fatal core-dumping signal, Valgrind will generate its own core file (`vgcore.NNNNN`) containing your program's state. You may use this core file for post-mortem debugging with GDB or similar. (Note: it will not generate a core if your core dump size limit is 0.) At the time of writing the core dumps do not include all the floating point register information.

In the unlikely event that Valgrind itself crashes, the operating system will create a core dump in the usual way.

2.10. Execution Trees

An execution tree (xtree) is made of a set of stack traces, each stack trace is associated with some resource consumptions or event counts. Depending on the xtree, different event counts/resource consumptions can be recorded in the xtree. Multiple tools can produce memory use xtree. Memcheck can output the leak search results in an xtree.

A typical usage for an xtree is to show a graphical or textual representation of the heap usage of a program. The below figure is a heap usage xtree graphical representation produced by kcachegrind. In the kcachegrind output, you can see that main current heap usage (allocated indirectly) is 528 bytes : 388 bytes allocated indirectly via a call to function f1 and 140 bytes indirectly allocated via a call to function f2. f2 has allocated memory by calling g2, while f1 has allocated memory by calling g11 and g12. g11, g12 and g2 have directly called a memory allocation function (malloc), and so have a non zero 'Self' value. Note that when kcachegrind shows an xtree, the 'Called' column and call nr indications in the Call Graph are not significant (always set to 0 or 1, independently of the real nr of calls. The kcachegrind versions >= 0.8.0 do not show anymore such irrelevant xtree call number information.

An xtree heap memory report is produced at the end of the execution when required using the option `--xtree-memory`. It can also be produced on demand using the `xtmemory` monitor command (see Valgrind monitor commands). Currently, an xtree heap memory report can be produced by the `memcheck`, `helgrind` and `massif` tools.

The xtrees produced by the option --xtree-memory or the `xtmemory` monitor command are showing the following events/resource consumption describing heap usage:

- `curB` current number of Bytes allocated. The number of allocated bytes is added to the `curB` value of a stack trace for each allocation. It is decreased when a block allocated by this stack trace is released (by another "freeing" stack trace)
- `curBk` current number of Blocks allocated, maintained similary to curB : +1 for each allocation, -1 when the block is freed.
- `totB` total allocated Bytes. This is increased for each allocation with the number of allocated bytes.
- `totBk` total allocated Blocks, maintained similary to totB : +1 for each allocation.
- `totFdB` total Freed Bytes, increased each time a block is released by this ("freeing") stack trace : + nr freed bytes for each free operation.
- `totFdBk` total Freed Blocks, maintained similarly to totFdB : +1 for each free operation.

Note that the last 4 counts are produced only when the `--xtree-memory=full` was given at startup.

Xtrees can be saved in 2 file formats, the "Callgrind Format" and the "Massif Format".

- Callgrind Format

  An xtree file in the Callgrind Format contains a single callgraph, associating each stack trace with the values recorded in the xtree.

  Different Callgrind Format file visualizers are available:

  Valgrind distribution includes the `callgrind_annotate` command line utility that reads in the xtree data, and prints a sorted lists of functions, optionally with source annotation. Note that due to xtree specificities, you must give the option `--inclusive=yes` to callgrind_annotate.

  For graphical visualization of the data, you can use KCachegrind, which is a KDE/Qt based GUI that makes it easy to navigate the large amount of data that an xtree can contain.

  Note that xtree Callgrind Format does not make use of the inline information even when specifying `--read-inline-info=yes`.

- Massif Format

  An xtree file in the Massif Format contains one detailed tree callgraph data for each type of event recorded in the xtree. So, for `--xtree-memory=alloc`, the output file will contain 2 detailed trees (for the counts `curB` and `curBk`), while `--xtree-memory=full` will give a file with 6 detailed trees.

  Different Massif Format file visualizers are available. Valgrind distribution includes the `ms_print` command line utility that produces an easy to read reprentation of a massif output file. See Using Massif and ms_print and Using massif-visualizer for more details about visualising Massif Format output files.

  Note that xtree Massif Format makes use of the inline information when specifying `--read-inline-info=yes`.

Note that for equivalent information, the Callgrind Format is more compact than the Massif Format. However, the Callgrind Format always contains the full data: there is no filtering done during file production, filtering is done by visualizers such as kcachegrind. kcachegrind is particularly easy to use to analyse big xtree data containing multiple events counts or resources consumption. The Massif Format (optionally) only contains a part of the data. For example, the Massif tool might filter some of the data, according to the `--threshold` option.

To clarify the xtree concept, the below gives several extracts of the output produced by the following commands:

```
valgrind --xtree-memory=full --xtree-memory-file=xtmemory.kcg mfg
callgrind_annotate --auto=yes --inclusive=yes --sort=curB:100,curBk:100,totB:100,totBk:100
```

The below extract shows that the program mfg has allocated in total 770 bytes in 60 different blocks. Of these 60 blocks, 19 were freed, releasing a total of 242 bytes. The heap currently contains 528 bytes in 41 blocks.

```
--------------------------------------------------------------------------------
curB curBk totB totBk totFdB totFdBk
--------------------------------------------------------------------------------
 528    41  770    60    242      19  PROGRAM TOTALS
```

The below gives more details about which functions have allocated or released memory. As an example, we see that main has (directly or indirectly) allocated 770 bytes of memory and freed (directly or indirectly) 242 bytes of memory.

The function f1 has (directly or indirectly) allocated 570 bytes of memory, and has not (directly or indirectly) freed memory. Of the 570 bytes allocated by function f1, 388 bytes (34 blocks) have not been released.

```
--------------------------------------------------------------------------------
curB curBk totB totBk totFdB totFdBk  file:function
--------------------------------------------------------------------------------
 528    41  770    60    242      19  mfg.c:main
 388    34  570    50      0       0  mfg.c:f1
 220    20  330    30      0       0  mfg.c:g11
 168    14  240    20      0       0  mfg.c:g12
 140     7  200    10      0       0  mfg.c:g2
 140     7  200    10      0       0  mfg.c:f2
   0     0    0     0    131      10  mfg.c:freeY
   0     0    0     0    111       9  mfg.c:freeX
```

The below gives a more detailed information about the callgraph and which source lines/calls have (directly or indirectly) allocated or released memory. The below shows that the 770 bytes allocated by main have been indirectly allocated by calls to f1 and f2. Similarly, we see that the 570 bytes allocated by f1 have been indirectly allocated by calls to g11 and g12. Of the 330 bytes allocated by the 30 calls to g11, 168 bytes have not been freed. The function freeY (called once by main) has released in total 10 blocks and 131 bytes.

```
--------------------------------------------------------------------------------
-- Auto-annotated source: /home/philippe/valgrind/littleprogs/ + mfg.c
--------------------------------------------------------------------------------
curB curBk totB totBk totFdB totFdBk
....
   .     .    .     .      .       .  static void freeY(void)
   .     .    .     .      .       .  {
   .     .    .     .      .       .     int i;
   .     .    .     .      .       .     for (i = 0; i < next_ptr; i++)
   .     .    .     .      .       .        if(i % 5 == 0 && ptrs[i] != NULL)
   0     0    0     0    131      10           free(ptrs[i]);
   .     .    .     .      .       .  }
   .     .    .     .      .       .  static void f1(void)
   .     .    .     .      .       .  {
   .     .    .     .      .       .     int i;
   .     .    .     .      .       .     for (i = 0; i < 30; i++)
 220    20  330    30      0       0        g11();
   .     .    .     .      .       .     for (i = 0; i < 20; i++)
 168    14  240    20      0       0        g12();
   .     .    .     .      .       .  }
   .     .    .     .      .       .  int main()
   .     .    .     .      .       .  {
 388    34  570    50      0       0     f1();
 140     7  200    10      0       0     f2();
   0     0    0     0    111       9     freeX();
   0     0    0     0    131      10     freeY();
   .     .    .     .      .       .     return 0;
   .     .    .     .      .       .  }
```

Heap memory xtrees are helping to understand how your (big) program is using the heap. A full heap memory xtree helps to pin point some code that allocates a lot of small objects : allocating such small objects might be replaced by more efficient technique, such as allocating a big block using malloc, and then diviving this block into smaller blocks in order to decrease the cpu and/or memory overhead of allocating a lot of small blocks. Such full xtree information complements e.g. what callgrind can show: callgrind can show the number of calls to a function (such as malloc) but does not indicate the volume of memory allocated (or freed).

A full heap memory xtree also can identify the code that allocates and frees a lot of blocks : the total foot print of the program might not reflect the fact that the same memory was over and over allocated then released.

Finally, Xtree visualizers such as kcachegrind are helping to identify big memory consumers, in order to possibly optimise the amount of memory needed by your program.

2.11. Building and Installing Valgrind

We use the standard Unix `./configure`, `make`, `make install` mechanism. Once you have completed `make install` you may then want to run the regression tests with `make regtest`.

In addition to the usual `--prefix=/path/to/install/tree`, there are three options which affect how Valgrind is built:

- `--enable-inner`

  This builds Valgrind with some special magic hacks which make it possible to run it on a standard build of Valgrind (what the developers call "self-hosting"). Ordinarily you should not use this option as various kinds of safety checks are disabled.

- `--enable-only64bit`

  `--enable-only32bit`

  On 64-bit platforms (amd64-linux, ppc64-linux, amd64-darwin), Valgrind is by default built in such a way that both 32-bit and 64-bit executables can be run. Sometimes this cleverness is a problem for a variety of reasons. These two options allow for single-target builds in this situation. If you issue both, the configure script will complain. Note they are ignored on 32-bit-only platforms (x86-linux, ppc32-linux, arm-linux, x86-darwin).

The `configure` script tests the version of the X server currently indicated by the current `$DISPLAY`. This is a known bug. The intention was to detect the version of the current X client libraries, so that correct suppressions could be selected for them, but instead the test checks the server version. This is just plain wrong.

If you are building a binary package of Valgrind for distribution, please read `README_PACKAGERS` Readme Packagers. It contains some important information.

Apart from that, there's not much excitement here. Let us know if you have build problems.

2.12. If You Have Problems

Contact us at http://www.valgrind.org/.

See Limitations for the known limitations of Valgrind, and for a list of programs which are known not to work on it.

All parts of the system make heavy use of assertions and internal self-checks. They are permanently enabled, and we have no plans to disable them. If one of them breaks, please mail us!

If you get an assertion failure in `m_mallocfree.c`, this may have happened because your program wrote off the end of a heap block, or before its beginning, thus corrupting heap metadata. Valgrind hopefully will have emitted a message to that effect before dying in this way.

Read the Valgrind FAQ for more advice about common problems, crashes, etc.

2.13. Limitations

The following list of limitations seems long. However, most programs actually work fine.

Valgrind will run programs on the supported platforms subject to the following constraints:

- On Linux, Valgrind determines at startup the size of the 'brk segment' using the RLIMIT_DATA rlim_cur, with a minimum of 1 MB and a maximum of 8 MB. Valgrind outputs a message each time a program tries to extend the brk segment beyond the size determined at startup. Most programs will work properly with this limit, typically by switching to the use of mmap to get more memory. If your program really needs a big brk segment, you must change the 8 MB hardcoded limit and recompile Valgrind.
- On x86 and amd64, there is no support for 3DNow! instructions. If the translator encounters these, Valgrind will generate a SIGILL when the instruction is executed. Apart from that, on x86 and amd64, essentially all instructions are supported, up to and including AVX and AES in 64-bit mode and SSSE3 in 32-bit mode. 32-bit mode does in fact support the bare minimum SSE4 instructions needed to run programs on MacOSX 10.6 on 32-bit targets.
- On ppc32 and ppc64, almost all integer, floating point and Altivec instructions are supported. Specifically: integer and FP insns that are mandatory for PowerPC, the "General-purpose optional" group (fsqrt, fsqrts, stfiwx), the "Graphics optional" group (fre, fres, frsqrte, frsqrtes), and the Altivec (also known as VMX) SIMD instruction set, are supported. Also, instructions from the Power ISA 2.05 specification, as present in POWER6 CPUs, are supported.
- On ARM, essentially the entire ARMv7-A instruction set is supported, in both ARM and Thumb mode. ThumbEE and Jazelle are not supported. NEON, VFPv3 and ARMv6 media support is fairly complete.
- If your program does its own memory management, rather than using malloc/new/free/delete, it should still work, but Memcheck's error checking won't be so effective. If you describe your program's memory management scheme using "client requests" (see The Client Request mechanism), Memcheck can do better. Nevertheless, using malloc/ new and free/delete is still the best approach.
- Valgrind's signal simulation is not as robust as it could be. Basic POSIX-compliant sigaction and sigprocmask functionality is supplied, but it's conceivable that things could go badly awry if you do weird things with signals. Workaround: don't. Programs that do non-POSIX signal tricks are in any case inherently unportable, so should be avoided if possible.
- Machine instructions, and system calls, have been implemented on demand. So it's possible, although unlikely, that a program will fall over with a message to that effect. If this happens, please report all the details printed out, so we can try and implement the missing feature.
- Memory consumption of your program is majorly increased whilst running under Valgrind's Memcheck tool. This is due to the large amount of administrative information maintained behind the scenes. Another cause is that Valgrind dynamically translates the original executable. Translated, instrumented code is 12-18 times larger than the original so you can easily end up with 150+ MB of translations when running (eg) a web browser.
- Valgrind can handle dynamically-generated code just fine. If you regenerate code over the top of old code (ie. at the same memory addresses), if the code is on the stack Valgrind will realise the code has changed, and work correctly. This is necessary to handle the trampolines GCC uses to implemented nested functions. If you regenerate code somewhere other than the stack, and you are running on an 32- or 64-bit x86 CPU, you will need to use the `--smc-check=all` option, and Valgrind will run more slowly than normal. Or you can add client requests that tell Valgrind when your program has overwritten code.

  On other platforms (ARM, PowerPC) Valgrind observes and honours the cache invalidation hints that programs are obliged to emit to notify new code, and so self-modifying-code support should work automatically, without the need for `--smc-check=all`.
- Valgrind has the following limitations in its implementation of x86/AMD64 floating point relative to IEEE754.

Precision: There is no support for 80 bit arithmetic. Internally, Valgrind represents all such "long double" numbers in 64 bits, and so there may be some differences in results. Whether or not this is critical remains to be seen. Note, the x86/amd64 fldt/fstpt instructions (read/write 80-bit numbers) are correctly simulated, using conversions to/from 64 bits, so that in-memory images of 80-bit numbers look correct if anyone wants to see.

The impression observed from many FP regression tests is that the accuracy differences aren't significant. Generally speaking, if a program relies on 80-bit precision, there may be difficulties porting it to non x86/amd64 platforms which only support 64-bit FP precision. Even on x86/amd64, the program may get different results depending on whether it is compiled to use SSE2 instructions (64-bits only), or x87 instructions (80-bit). The net effect is to make FP programs behave as if they had been run on a machine with 64-bit IEEE floats, for example PowerPC. On amd64 FP arithmetic is done by default on SSE2, so amd64 looks more like PowerPC than x86 from an FP perspective, and there are far fewer noticeable accuracy differences than with x86.

Rounding: Valgrind does observe the 4 IEEE-mandated rounding modes (to nearest, to +infinity, to -infinity, to zero) for the following conversions: float to integer, integer to float where there is a possibility of loss of precision, and float-to-float rounding. For all other FP operations, only the IEEE default mode (round to nearest) is supported.

Numeric exceptions in FP code: IEEE754 defines five types of numeric exception that can happen: invalid operation (sqrt of negative number, etc), division by zero, overflow, underflow, inexact (loss of precision).

For each exception, two courses of action are defined by IEEE754: either (1) a user-defined exception handler may be called, or (2) a default action is defined, which "fixes things up" and allows the computation to proceed without throwing an exception.

Currently Valgrind only supports the default fixup actions. Again, feedback on the importance of exception support would be appreciated.

When Valgrind detects that the program is trying to exceed any of these limitations (setting exception handlers, rounding mode, or precision control), it can print a message giving a traceback of where this has happened, and continue execution. This behaviour used to be the default, but the messages are annoying and so showing them is now disabled by default. Use `--show-emwarns=yes` to see them.

The above limitations define precisely the IEEE754 'default' behaviour: default fixup on all exceptions, round-to-nearest operations, and 64-bit precision.

- Valgrind has the following limitations in its implementation of x86/AMD64 SSE2 FP arithmetic, relative to IEEE754.

  Essentially the same: no exceptions, and limited observance of rounding mode. Also, SSE2 has control bits which make it treat denormalised numbers as zero (DAZ) and a related action, flush denormals to zero (FTZ). Both of these cause SSE2 arithmetic to be less accurate than IEEE requires. Valgrind detects, ignores, and can warn about, attempts to enable either mode.

- Valgrind has the following limitations in its implementation of ARM VFPv3 arithmetic, relative to IEEE754.

  Essentially the same: no exceptions, and limited observance of rounding mode. Also, switching the VFP unit into vector mode will cause Valgrind to abort the program -- it has no way to emulate vector uses of VFP at a reasonable performance level. This is no big deal given that non-scalar uses of VFP instructions are in any case deprecated.

- Valgrind has the following limitations in its implementation of PPC32 and PPC64 floating point arithmetic, relative to IEEE754.

  Scalar (non-Altivec): Valgrind provides a bit-exact emulation of all floating point instructions, except for "fre" and "fres", which are done more precisely than required by the PowerPC architecture specification. All floating point operations observe the current rounding mode.

However, fpscr[FPRF] is not set after each operation. That could be done but would give measurable performance overheads, and so far no need for it has been found.

As on x86/AMD64, IEEE754 exceptions are not supported: all floating point exceptions are handled using the default IEEE fixup actions. Valgrind detects, ignores, and can warn about, attempts to unmask the 5 IEEE FP exception kinds by writing to the floating-point status and control register (fpscr).

Vector (Altivec, VMX): essentially as with x86/AMD64 SSE/SSE2: no exceptions, and limited observance of rounding mode. For Altivec, FP arithmetic is done in IEEE/Java mode, which is more accurate than the Linux default setting. "More accurate" means that denormals are handled properly, rather than simply being flushed to zero.

Programs which are known not to work are:

- emacs starts up but immediately concludes it is out of memory and aborts. It may be that Memcheck does not provide a good enough emulation of the `mallinfo` function. Emacs works fine if you build it to use the standard malloc/free routines.

2.14. An Example Run

This is the log for a run of a small program using Memcheck. The program is in fact correct, and the reported error is as the result of a potentially serious code generation bug in GNU g++ (snapshot 20010527).

```
sewardj@phoenix:~/newmat10$ ~/Valgrind-6/valgrind -v ./bogon
==25832== Valgrind 0.10, a memory error detector for x86 RedHat 7.1.
==25832== Copyright (C) 2000-2001, and GNU GPL'd, by Julian Seward.
==25832== Startup, with flags:
==25832== --suppressions=/home/sewardj/Valgrind/redhat71.supp
==25832== reading syms from /lib/ld-linux.so.2
==25832== reading syms from /lib/libc.so.6
==25832== reading syms from /mnt/pima/jrs/Inst/lib/libgcc_s.so.0
==25832== reading syms from /lib/libm.so.6
==25832== reading syms from /mnt/pima/jrs/Inst/lib/libstdc++.so.3
==25832== reading syms from /home/sewardj/Valgrind/valgrind.so
==25832== reading syms from /proc/self/exe
==25832==
==25832== Invalid read of size 4
==25832==    at 0x8048724: BandMatrix::ReSize(int,int,int) (bogon.cpp:45)
==25832==    by 0x80487AF: main (bogon.cpp:66)
==25832==  Address 0xBFFFF74C is not stack'd, malloc'd or free'd
==25832==
==25832== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==25832== malloc/free: in use at exit: 0 bytes in 0 blocks.
==25832== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==25832== For a detailed leak analysis, rerun with: --leak-check=yes
```

The GCC folks fixed this about a week before GCC 3.0 shipped.

2.15. Warning Messages You Might See

Some of these only appear if you run in verbose mode (enabled by `-v`):

- `More than 100 errors detected. Subsequent errors will still be recorded, but in less detail than before.`

After 100 different errors have been shown, Valgrind becomes more conservative about collecting them. It then requires only the program counters in the top two stack frames to match when deciding whether or not two errors are really the same one. Prior to this point, the PCs in the top four frames are required to match. This hack has the effect of slowing down the appearance of new errors after the first 100. The 100 constant can be changed by recompiling Valgrind.

- `More than 1000 errors detected. I'm not reporting any more. Final error counts may be inaccurate. Go fix your program!`

  After 1000 different errors have been detected, Valgrind ignores any more. It seems unlikely that collecting even more different ones would be of practical help to anybody, and it avoids the danger that Valgrind spends more and more of its time comparing new errors against an ever-growing collection. As above, the 1000 number is a compile-time constant.

- `Warning: client switching stacks?`

  Valgrind spotted such a large change in the stack pointer that it guesses the client is switching to a different stack. At this point it makes a kludgey guess where the base of the new stack is, and sets memory permissions accordingly. At the moment "large change" is defined as a change of more that 2000000 in the value of the stack pointer register. If Valgrind guesses wrong, you may get many bogus error messages following this and/or have crashes in the stack trace recording code. You might avoid these problems by informing Valgrind about the stack bounds using VALGRIND_STACK_REGISTER client request.

- `Warning: client attempted to close Valgrind's logfile fd <number>`

  Valgrind doesn't allow the client to close the logfile, because you'd never see any diagnostic information after that point. If you see this message, you may want to use the `--log-fd=<number>` option to specify a different logfile file-descriptor number.

- `Warning: noted but unhandled ioctl <number>`

  Valgrind observed a call to one of the vast family of `ioctl` system calls, but did not modify its memory status info (because nobody has yet written a suitable wrapper). The call will still have gone through, but you may get spurious errors after this as a result of the non-update of the memory info.

- `Warning: set address range perms: large range <number>`

  Diagnostic message, mostly for benefit of the Valgrind developers, to do with memory permissions.

3. Using and understanding the Valgrind core: Advanced Topics

This chapter describes advanced aspects of the Valgrind core services, which are mostly of interest to power users who wish to customise and modify Valgrind's default behaviours in certain useful ways. The subjects covered are:

- The "Client Request" mechanism
- Debugging your program using Valgrind's gdbserver and GDB
- Function Wrapping

3.1. The Client Request mechanism

Valgrind has a trapdoor mechanism via which the client program can pass all manner of requests and queries to Valgrind and the current tool. Internally, this is used extensively to make various things work, although that's not visible from the outside.

For your convenience, a subset of these so-called client requests is provided to allow you to tell Valgrind facts about the behaviour of your program, and also to make queries. In particular, your program can tell Valgrind about things that it otherwise would not know, leading to better results.

Clients need to include a header file to make this work. Which header file depends on which client requests you use. Some client requests are handled by the core, and are defined in the header file `valgrind/valgrind.h`. Tool-specific header files are named after the tool, e.g. `valgrind/memcheck.h`. Each tool-specific header file includes `valgrind/valgrind.h` so you don't need to include it in your client if you include a tool-specific header. All header files can be found in the `include/valgrind` directory of wherever Valgrind was installed.

The macros in these header files have the magical property that they generate code in-line which Valgrind can spot. However, the code does nothing when not run on Valgrind, so you are not forced to run your program under Valgrind just because you use the macros in this file. Also, you are not required to link your program with any extra supporting libraries.

The code added to your binary has negligible performance impact: on x86, amd64, ppc32, ppc64 and ARM, the overhead is 6 simple integer instructions and is probably undetectable except in tight loops. However, if you really wish to compile out the client requests, you can compile with `-DNVALGRIND` (analogous to `-DNDEBUG`'s effect on `assert`).

You are encouraged to copy the `valgrind/*.h` headers into your project's include directory, so your program doesn't have a compile-time dependency on Valgrind being installed. The Valgrind headers, unlike most of the rest of the code, are under a BSD-style license so you may include them without worrying about license incompatibility.

Here is a brief description of the macros available in `valgrind.h`, which work with more than one tool (see the tool-specific documentation for explanations of the tool-specific macros).

`RUNNING_ON_VALGRIND`:

Returns 1 if running on Valgrind, 0 if running on the real CPU. If you are running Valgrind on itself, returns the number of layers of Valgrind emulation you're running on.

`VALGRIND_DISCARD_TRANSLATIONS:`

Discards translations of code in the specified address range. Useful if you are debugging a JIT compiler or some other dynamic code generation system. After this call, attempts to execute code in the invalidated address range will cause Valgrind to make new translations of that code, which is probably the semantics you want. Note that code invalidations are expensive because finding all the relevant translations quickly is very difficult, so try not to call it often. Note that you can be clever about this: you only need to call it when an area which previously contained code is overwritten with new code. You can choose to write code into fresh memory, and just call this occasionally to discard large chunks of old code all at once.

Alternatively, for transparent self-modifying-code support, use`--smc-check=all`, or run on ppc32/Linux, ppc64/Linux or ARM/Linux.

`VALGRIND_COUNT_ERRORS`:

Returns the number of errors found so far by Valgrind. Can be useful in test harness code when combined with the `--log-fd=-1` option; this runs Valgrind silently, but the client program can detect when errors occur. Only useful for tools that report errors, e.g. it's useful for Memcheck, but for Cachegrind it will always return zero because Cachegrind doesn't report errors.

`VALGRIND_MALLOCLIKE_BLOCK`:

If your program manages its own memory instead of using the standard `malloc` / `new` / `new[]`, tools that track information about heap blocks will not do nearly as good a job. For example, Memcheck won't detect nearly as many errors, and the error messages won't be as informative. To improve this situation, use this macro just after your custom allocator allocates some new memory. See the comments in `valgrind.h` for information on how to use it.

`VALGRIND_FREELIKE_BLOCK`:

This should be used in conjunction with `VALGRIND_MALLOCLIKE_BLOCK`. Again, see `valgrind.h` for information on how to use it.

`VALGRIND_RESIZEINPLACE_BLOCK`:

Informs a Valgrind tool that the size of an allocated block has been modified but not its address. See `valgrind.h` for more information on how to use it.

`VALGRIND_CREATE_MEMPOOL`, `VALGRIND_DESTROY_MEMPOOL`, `VALGRIND_MEMPOOL_ALLOC`, `VALGRIND_MEMPOOL_FREE`, `VALGRIND_MOVE_MEMPOOL`, `VALGRIND_MEMPOOL_CHANGE`, `VALGRIND_MEMPOOL_EXISTS`:

These are similar to `VALGRIND_MALLOCLIKE_BLOCK` and `VALGRIND_FREELIKE_BLOCK` but are tailored towards code that uses memory pools. See Memory Pools for a detailed description.

`VALGRIND_NON_SIMD_CALL[0123]`:

Executes a function in the client program on the *real* CPU, not the virtual CPU that Valgrind normally runs code on. The function must take an integer (holding a thread ID) as the first argument and then 0, 1, 2 or 3 more arguments (depending on which client request is used). These are used in various ways internally to Valgrind. They might be useful to client programs.

Warning: Only use these if you *really* know what you are doing. They aren't entirely reliable, and can cause Valgrind to crash. See `valgrind.h` for more details.

`VALGRIND_PRINTF(format, ...)`:

Print a printf-style message to the Valgrind log file. The message is prefixed with the PID between a pair of `**` markers. (Like all client requests, nothing is output if the client program is not running under Valgrind.) Output is not produced until a newline is encountered, or subsequent Valgrind output is printed; this allows you to build up a single line of output over multiple calls. Returns the number of characters output, excluding the PID prefix.

`VALGRIND_PRINTF_BACKTRACE(format, ...):`

Like `VALGRIND_PRINTF` (in particular, the return value is identical), but prints a stack backtrace immediately afterwards.

`VALGRIND_MONITOR_COMMAND(command):`

Execute the given monitor command (a string). Returns 0 if command is recognised. Returns 1 if command is not recognised. Note that some monitor commands provide access to a functionality also accessible via a specific client request. For example, memcheck leak search can be requested from the client program using VALGRIND_DO_LEAK_CHECK or via the monitor command "leak_search". Note that the syntax of the command string is only verified at run-time. So, if it exists, it is preferable to use a specific client request to have better compile time verifications of the arguments.

`VALGRIND_CLO_CHANGE(option):`

Changes the value of a dynamically changeable option (a string). See Dynamically Change Options.

`VALGRIND_STACK_REGISTER(start, end):`

Registers a new stack. Informs Valgrind that the memory range between start and end is a unique stack. Returns a stack identifier that can be used with other `VALGRIND_STACK_*` calls.

Valgrind will use this information to determine if a change to the stack pointer is an item pushed onto the stack or a change over to a new stack. Use this if you're using a user-level thread package and are noticing crashes in stack trace recording or spurious errors from Valgrind about uninitialized memory reads.

Warning: Unfortunately, this client request is unreliable and best avoided.

`VALGRIND_STACK_DEREGISTER(id):`

Deregisters a previously registered stack. Informs Valgrind that previously registered memory range with stack id `id` is no longer a stack.

Warning: Unfortunately, this client request is unreliable and best avoided.

`VALGRIND_STACK_CHANGE(id, start, end):`

Changes a previously registered stack. Informs Valgrind that the previously registered stack with stack id `id` has changed its start and end values. Use this if your user-level thread package implements stack growth.

Warning: Unfortunately, this client request is unreliable and best avoided.

3.2. Debugging your program using Valgrind gdbserver and GDB

A program running under Valgrind is not executed directly by the CPU. Instead it runs on a synthetic CPU provided by Valgrind. This is why a debugger cannot natively debug your program when it runs on Valgrind.

This section describes how GDB can interact with the Valgrind gdbserver to provide a fully debuggable program under Valgrind. Used in this way, GDB also provides an interactive usage of Valgrind core or tool functionalities, including incremental leak search under Memcheck and on-demand Massif snapshot production.

3.2.1. Quick Start: debugging in 3 steps

The simplest way to get started is to run Valgrind with the flag `--vgdb-error=0`. Then follow the on-screen directions, which give you the precise commands needed to start GDB and connect it to your program.

Otherwise, here's a slightly more verbose overview.

If you want to debug a program with GDB when using the Memcheck tool, start Valgrind like this:

```
valgrind --vgdb=yes --vgdb-error=0 prog
```

In another shell, start GDB:

```
gdb prog
```

Then give the following command to GDB:

```
(gdb) target remote | vgdb
```

You can now debug your program e.g. by inserting a breakpoint and then using the GDB `continue` command.

This quick start information is enough for basic usage of the Valgrind gdbserver. The sections below describe more advanced functionality provided by the combination of Valgrind and GDB. Note that the command line flag `--vgdb=yes` can be omitted, as this is the default value.

3.2.2. Valgrind gdbserver overall organisation

The GNU GDB debugger is typically used to debug a process running on the same machine. In this mode, GDB uses system calls to control and query the program being debugged. This works well, but only allows GDB to debug a program running on the same computer.

GDB can also debug processes running on a different computer. To achieve this, GDB defines a protocol (that is, a set of query and reply packets) that facilitates fetching the value of memory or registers, setting breakpoints, etc. A gdbserver is an implementation of this "GDB remote debugging" protocol. To debug a process running on a remote computer, a gdbserver (sometimes called a GDB stub) must run at the remote computer side.

The Valgrind core provides a built-in gdbserver implementation, which is activated using `--vgdb=yes` or `--vgdb=full`. This gdbserver allows the process running on Valgrind's synthetic CPU to be debugged remotely. GDB sends protocol query packets (such as "get register contents") to the Valgrind embedded gdbserver. The gdbserver executes the queries (for example, it will get the register values of the synthetic CPU) and gives the results back to GDB.

GDB can use various kinds of channels (TCP/IP, serial line, etc) to communicate with the gdbserver. In the case of Valgrind's gdbserver, communication is done via a pipe and a small helper program called vgdb, which acts as an intermediary. If no GDB is in use, vgdb can also be used to send monitor commands to the Valgrind gdbserver from a shell command line.

3.2.3. Connecting GDB to a Valgrind gdbserver

To debug a program "`prog`" running under Valgrind, you must ensure that the Valgrind gdbserver is activated by specifying either `--vgdb=yes` or `--vgdb=full`. A secondary command line option, `--vgdb-error=number`, can be used to tell the gdbserver only to become active once the specified number of errors have been shown. A value of zero will therefore cause the gdbserver to become active at startup, which allows you to insert breakpoints before starting the run. For example:

```
valgrind --tool=memcheck --vgdb=yes --vgdb-error=0 ./prog
```

The Valgrind gdbserver is invoked at startup and indicates it is waiting for a connection from a GDB:

```
==2418== Memcheck, a memory error detector
==2418== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2418== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==2418== Command: ./prog
==2418==
==2418== (action at startup) vgdb me ...
```

GDB (in another shell) can then be connected to the Valgrind gdbserver. For this, GDB must be started on the program `prog`:

```
gdb ./prog
```

You then indicate to GDB that you want to debug a remote target:

```
(gdb) target remote | vgdb
```

GDB then starts a vgdb relay application to communicate with the Valgrind embedded gdbserver:

```
(gdb) target remote | vgdb
Remote debugging using | vgdb
relaying data between gdb and process 2418
Reading symbols from /lib/ld-linux.so.2...done.
Reading symbols from /usr/lib/debug/lib/ld-2.11.2.so.debug...done.
Loaded symbols for /lib/ld-linux.so.2
[Switching to Thread 2418]
0x001f2850 in _start () from /lib/ld-linux.so.2
(gdb)
```

Note that vgdb is provided as part of the Valgrind distribution. You do not need to install it separately.

If vgdb detects that there are multiple Valgrind gdbservers that can be connected to, it will list all such servers and their PIDs, and then exit. You can then reissue the GDB "target" command, but specifying the PID of the process you want to debug:

```
(gdb) target remote | vgdb
Remote debugging using | vgdb
no --pid= arg given and multiple valgrind pids found:
use --pid=2479 for valgrind --tool=memcheck --vgdb=yes --vgdb-error=0 ./prog
use --pid=2481 for valgrind --tool=memcheck --vgdb=yes --vgdb-error=0 ./prog
use --pid=2483 for valgrind --vgdb=yes --vgdb-error=0 ./another_prog
Remote communication error: Resource temporarily unavailable.
(gdb)  target remote | vgdb --pid=2479
Remote debugging using | vgdb --pid=2479
relaying data between gdb and process 2479
Reading symbols from /lib/ld-linux.so.2...done.
Reading symbols from /usr/lib/debug/lib/ld-2.11.2.so.debug...done.
Loaded symbols for /lib/ld-linux.so.2
[Switching to Thread 2479]
0x001f2850 in _start () from /lib/ld-linux.so.2
(gdb)
```

If you want to use the `--multi` mode which makes vgdb start in extended-remote mode, set the following in GDB:

```
# gdb prog
(gdb) set remote exec-file prog
(gdb) set sysroot /
(gdb) target extended-remote | vgdb --multi --vargs -q
(gdb) start
Temporary breakpoint 1 at 0x24e0
Starting program: prog
relaying data between gdb and process 2999348

Temporary breakpoint 1, 0x000000000010a4a0 in main ()
(gdb)
```

Note that in `--multi` mode you don't have to start valgrind separately. vgdb will start valgrind for you. vgdb `--multi` mode is experimental and currently has some limitations like not being able to see program stdin and stdout. Also you have to explicitly set the remote exec-file and sysroot to tell GDB the "remote" and local files are the same.

Once GDB is connected to the Valgrind gdbserver, it can be used in the same way as if you were debugging the program natively:

- Breakpoints can be inserted or deleted.
- Variables and register values can be examined or modified.
- Signal handling can be configured (printing, ignoring).
- Execution can be controlled (continue, step, next, stepi, etc).
- Program execution can be interrupted using Control-C.

And so on. Refer to the GDB user manual for a complete description of GDB's functionality.

3.2.4. Connecting to an Android gdbserver

When developping applications for Android, you will typically use a development system (on which the Android NDK is installed) to compile your application. An Android target system or emulator will be used to run the application. In this setup, Valgrind and vgdb will run on the Android system, while GDB will run on the development system. GDB will connect to the vgdb running on the Android system using the Android NDK 'adb forward' application.

Example: on the Android system, execute the following:

```
valgrind --vgdb-error=0 --vgdb=yes prog
# and then in another shell, run:
vgdb --port=1234
```

On the development system, execute the following commands:

```
adb forward tcp:1234 tcp:1234
gdb prog
(gdb) target remote :1234
```

GDB will use a local tcp/ip connection to connect to the Android adb forwarder. Adb will establish a relay connection between the host system and the Android target system. Be sure to use the GDB delivered in the Android NDK system (typically, arm-linux-androideabi-gdb), as the host GDB is probably not able to debug Android arm applications. Note that the local port nr (used by GDB) must not necessarily be equal to the port number used by vgdb: adb can forward tcp/ip between different port numbers.

In the current release, the GDB server is not enabled by default for Android, due to problems in establishing a suitable directory in which Valgrind can create the necessary FIFOs (named pipes) for communication purposes. You can stil try to use the GDB server, but you will need to explicitly enable it using the flag `--vgdb=yes` or `--vgdb=full`.

Additionally, you will need to select a temporary directory which is (a) writable by Valgrind, and (b) supports FIFOs. This is the main difficult point. Often, `/sdcard` satisfies requirement (a), but fails for (b) because it is a VFAT file system and VFAT does not support pipes. Possibilities you could try are `/data/local`, `/data/local/Inst` (if you installed Valgrind there), or `/data/data/name.of.my.app`, if you are running a specific application and it has its own directory of that form. This last possibility may have the highest probability of success.

You can specify the temporary directory to use either via the `--with-tmpdir=` configure time flag, or by setting environment variable TMPDIR when running Valgrind (on the Android device, not on the Android NDK development host). Another alternative is to specify the directory for the FIFOs using the `--vgdb-prefix=` Valgrind command line option.

We hope to have a better story for temporary directory handling on Android in the future. The difficulty is that, unlike in standard Unixes, there is no single temporary file directory that reliably works across all devices and scenarios.

3.2.5. Monitor command handling by the Valgrind gdbserver

The Valgrind gdbserver provides additional Valgrind-specific functionality via "monitor commands". Such monitor commands can be sent from the GDB command line or from the shell command line or requested by the client program using the VALGRIND_MONITOR_COMMAND client request. See Valgrind monitor commands for the list of the Valgrind core monitor commands available regardless of the Valgrind tool selected.

The following tools provide tool-specific monitor commands:

- Memcheck Monitor Commands
- Callgrind Monitor Commands
- Massif Monitor Commands
- Helgrind Monitor Commands

An example of a tool specific monitor command is the Memcheck monitor command `leak_check full reachable any`. This requests a full reporting of the allocated memory blocks. To have this leak check executed, use the GDB command:

```
(gdb) monitor leak_check full reachable any
```

GDB sends as a single string all what follows 'monitor' to the Valgrind gdbserver. The Valgrind gdbserver parses the string and will execute the monitor command itself, if it recognises it to be a Valgrind core monitor command. If it is not recognised as such, it is assumed to be tool-specific and is handed to the tool for execution. For example:

```
(gdb) monitor leak_check full reachable any
==2418== 100 bytes in 1 blocks are still reachable in loss record 1 of 1
==2418==    at 0x4006E9E: malloc (vg_replace_malloc.c:236)
==2418==    by 0x804884F: main (prog.c:88)
==2418==
==2418== LEAK SUMMARY:
```

```
==2418==    definitely lost: 0 bytes in 0 blocks
==2418==    indirectly lost: 0 bytes in 0 blocks
==2418==      possibly lost: 0 bytes in 0 blocks
==2418==    still reachable: 100 bytes in 1 blocks
==2418==         suppressed: 0 bytes in 0 blocks
==2418==
(gdb)
```

Similarly to GDB, the Valgrind gdbserver will accept abbreviated monitor command names and arguments, as long as the given abbreviation is unambiguous. For example, the above `leak_check` command can also be typed as:

```
(gdb) mo l f r a
```

The letters `mo` are recognised by GDB as being an abbreviation for `monitor`. So GDB sends the string `l f r a` to the Valgrind gdbserver. The letters provided in this string are unambiguous for the Valgrind gdbserver. This therefore gives the same output as the unabbreviated command and arguments. If the provided abbreviation is ambiguous, the Valgrind gdbserver will report the list of commands (or argument values) that can match:

```
(gdb) mo v. n
v. can match v.set v.info v.wait v.kill v.translate v.do
(gdb) mo v.i n
n_errs_found 0 n_errs_shown 0 (vgdb-error 0)
(gdb)
```

Instead of sending a monitor command from GDB, you can also send these from a shell command line. For example, the following command lines, when given in a shell, will cause the same leak search to be executed by the process 3145:

```
vgdb --pid=3145 leak_check full reachable any
vgdb --pid=3145 l f r a
```

Note that the Valgrind gdbserver automatically continues the execution of the program after a standalone invocation of vgdb. Monitor commands sent from GDB do not cause the program to continue: the program execution is controlled explicitly using GDB commands such as "continue" or "next".

Many monitor commands (e.g. v.info location, memcheck who_points_at, ...) require an address argument and an optional length: `<addr> [<len>]`. The arguments can also be provided by using a 'C array like syntax' by providing the address followed by the length between square brackets.

For example, the following two monitor commands provide the same information:

```
(gdb) mo xb 0x804a2f0 10
...
(gdb) mo xb 0x804a2f0[10]
...
```

3.2.6. GDB front end commands for Valgrind gdbserver monitor commands

As explained in Monitor command handling by the Valgrind gdbserver, valgrind monitor commands consist in strings that are not interpreted by GDB. GDB has no knowledge of these valgrind monitor commands. The GDB 'command line interface' infrastructure however provides interesting functionalities to help typing commands such as auto-completion, command specific help, searching for a command or command help matching a regexp, ...

To have a better integration of the valgrind monitor commands in the GDB command line interface, Valgrind provides python code defining a GDB front end command for each valgrind monitor command. Similarly, for each tool specific monitor command, the python code provides a matching GDB front end command.

Like other GDB commands, the GDB front end Valgrind monitor commands are hierarchically structured starting from 5 "top" GDB commands. Subcommands are defined below these "top" commands. To ease typing, shorter aliases are also provided.

- `valgrind` (aliased by `vg` and `v`) is the top GDB command providing front end commands to the Valgrind general monitor commands.
- `memcheck` (aliased by `mc`) is the top GDB command providing the front end commands corresponding to the memcheck specific monitor commands.
- `callgrind` (aliased by `cg`) is the top GDB command providing the front end commands corresponding to the callgrind specific monitor commands.
- `massif` (aliased by `ms`) is the top GDB command providing the front end commands corresponding to the massif specific monitor commands.
- `helgrind` (aliased by `hg`) is the top GDB command providing the front end commands corresponding to the helgrind specific monitor commands.

The usage of a GDB front end command is compatible with a direct usage of the Valgrind monitor command. The below example shows a direct usage of the Memcheck monitor command `xb` to examine the definedness status of the some_mem array and equivalent usages based on the GDB front end commands.

```
(gdb) list
1 int main()
2 {
3   char some_mem[5];
4   return 0;
5 }
(gdb) p &some_mem
$2 = (char (*)[5]) 0x1ffefffe5b
(gdb) p sizeof(some_mem)
$3 = 5
(gdb) monitor xb 0x1ffefffe5b 5
    ff    ff    ff    ff    ff
0x1FFEFFFE5B: 0x00 0x00 0x00 0x00 0x00
(gdb) memcheck xb 0x1ffefffe5b 5
    ff    ff    ff    ff    ff
0x1FFEFFFE5B: 0x00 0x00 0x00 0x00 0x00
(gdb) mc xb &some_mem sizeof(some_mem)
    ff    ff    ff    ff    ff
0x1FFEFFFE5B: 0x00 0x00 0x00 0x00 0x00
(gdb)
```

It is worth noting down that the third command uses the alias `mc`. This command also shows a significant advantage of using the GDB front end commands: as GDB "understands" the structure of these front end commands, where relevant, these front end commands will evaluate their arguments. In the case of the `xb` command, the GDB `xb` command evaluates its second argument (which must be an address expression) and its optional second argument (which must be an integer expression).

GDB will auto-load the python code defining the Valgrind front end commands as soon as GDB detects that the executable being debugged is running under valgrind. This detection is based on observing that the Valgrind process has loaded a specific Valgrind shared library. The loading of this library is done by the dynamic loader very early on in the execution of the process. If GDB is used to connect to a Valgrind process that has not yet started its execution (such as when Valgrind was started with the option `--vgdb-stop-at=startup` or `--vgdb-error=0`), then the GDB front end commands will not yet be auto-loaded. To have the GDB front end commands auto-loaded, you can put a breakpoint e.g. in main and use the GDB command `continue`. Alternatively, you can add in your .gdbinit a line that loads the python code at GDB startup such as:

```
source /path/to/valgrind/python/code/valgrind-monitor.py
```

The exact path to use in the source command depends on your Valgrind installation. The output of the shell command `vgdb --help` contains the absolute path name for the python file you can source in your .gdbinit to define the GDB valgrind front end monitor commands.

3.2.7. Valgrind gdbserver thread information

Valgrind's gdbserver enriches the output of the GDB `info threads` command with Valgrind-specific information. The operating system's thread number is followed by Valgrind's internal index for that thread ("tid") and by the Valgrind scheduler thread state:

```
(gdb) info threads
  4 Thread 6239 (tid 4 VgTs_Yielding)  0x001f2832 in _dl_sysinfo_int80 () from /lib/ld-lin
* 3 Thread 6238 (tid 3 VgTs_Runnable)  make_error (s=0x8048b76 "called from London") at pr
  2 Thread 6237 (tid 2 VgTs_WaitSys)  0x001f2832 in _dl_sysinfo_int80 () from /lib/ld-linu
  1 Thread 6234 (tid 1 VgTs_Yielding)  main (argc=1, argv=0xbedcc274) at prog.c:105
(gdb)
```

3.2.8. Examining and modifying Valgrind shadow registers

When the option `--vgdb-shadow-registers=yes` is given, the Valgrind gdbserver will let GDB examine and/or modify Valgrind's shadow registers. GDB version 7.1 or later is needed for this to work. For x86 and amd64, GDB version 7.2 or later is needed.

For each CPU register, the Valgrind core maintains two shadow register sets. These shadow registers can be accessed from GDB by giving a postfix `s1` or `s2` for respectively the first and second shadow register. For example, the x86 register `eax` and its two shadows can be examined using the following commands:

```
(gdb) p $eax
$1 = 0
(gdb) p $eaxs1
$2 = 0
(gdb) p $eaxs2
$3 = 0
(gdb)
```

Float shadow registers are shown by GDB as unsigned integer values instead of float values, as it is expected that these shadow values are mostly used for memcheck validity bits.

Intel/amd64 AVX registers `ymm0` to `ymm15` have also their shadow registers. However, GDB presents the shadow values using two "half" registers. For example, the half shadow registers for `ymm9` are `xmm9s1` (lower half for set 1), `ymm9hs1` (upper half for set 1), `xmm9s2` (lower half for set 2), `ymm9hs2` (upper half for set 2). Note the inconsistent notation for the names of the half registers: the lower part starts with an `x`, the upper part starts with an `y` and has an `h` before the shadow postfix.

The special presentation of the AVX shadow registers is due to the fact that GDB independently retrieves the lower and upper half of the `ymm` registers. GDB does not however know that the shadow half registers have to be shown combined.

3.2.9. Limitations of the Valgrind gdbserver

Debugging with the Valgrind gdbserver is very similar to native debugging. Valgrind's gdbserver implementation is quite complete, and so provides most of the GDB debugging functionality. There are however some limitations and peculiarities:

- Precision of "stop-at" commands.

  GDB commands such as "step", "next", "stepi", breakpoints and watchpoints, will stop the execution of the process. With the option `--vgdb=yes`, the process might not stop at the exact requested instruction. Instead, it might continue execution of the current basic block and stop at one of the following basic blocks. This is linked to the fact that Valgrind gdbserver has to instrument a block to allow stopping at the exact instruction requested. Currently, re-instrumentation of the block currently being executed is not supported. So, if the action requested by GDB (e.g. single stepping or inserting a breakpoint) implies re-instrumentation of the current block, the GDB action may not be executed precisely.

  This limitation applies when the basic block currently being executed has not yet been instrumented for debugging. This typically happens when the gdbserver is activated due to the tool reporting an error or to a watchpoint. If the gdbserver block has been activated following a breakpoint, or if a breakpoint has been inserted in the block before its execution, then the block has already been instrumented for debugging.

  If you use the option `--vgdb=full`, then GDB "stop-at" commands will be obeyed precisely. The downside is that this requires each instruction to be instrumented with an additional call to a gdbserver helper function, which gives considerable overhead (+500% for memcheck) compared to `--vgdb=no`. Option `--vgdb=yes` has neglectible overhead compared to `--vgdb=no`.

- Processor registers and flags values.

  When Valgrind gdbserver stops on an error, on a breakpoint or when single stepping, registers and flags values might not be always up to date due to the optimisations done by the Valgrind core. The default value `--vex-iropt-register-updates=unwindregs-at-mem-access` ensures that the registers needed to make a stack trace (typically PC/SP/FP) are up to date at each memory access (i.e. memory exception points). Disabling some optimisations using the following values will increase the precision of registers and flags values (a typical performance impact for memcheck is given for each option).

  - `--vex-iropt-register-updates=allregs-at-mem-access` (+10%) ensures that all registers and flags are up to date at each memory access.

  - `--vex-iropt-register-updates=allregs-at-each-insn` (+25%) ensures that all registers and flags are up to date at each instruction.

  Note that `--vgdb=full` (+500%, see above Precision of "stop-at" commands) automatically activates `--vex-iropt-register-updates=allregs-at-each-insn`.

- Hardware watchpoint support by the Valgrind gdbserver.

  The Valgrind gdbserver can simulate hardware watchpoints if the selected tool provides support for it. Currently, only Memcheck provides hardware watchpoint simulation. The hardware watchpoint simulation provided by Memcheck is much faster that GDB software watchpoints, which are implemented by GDB checking the value of the watched zone(s) after each instruction. Hardware watchpoint simulation also provides read watchpoints.

The hardware watchpoint simulation by Memcheck has some limitations compared to real hardware watchpoints. However, the number and length of simulated watchpoints are not limited.

Typically, the number of (real) hardware watchpoints is limited. For example, the x86 architecture supports a maximum of 4 hardware watchpoints, each watchpoint watching 1, 2, 4 or 8 bytes. The Valgrind gdbserver does not have any limitation on the number of simulated hardware watchpoints. It also has no limitation on the length of the memory zone being watched. Using GDB version 7.4 or later allow full use of the flexibility of the Valgrind gdbserver's simulated hardware watchpoints. Previous GDB versions do not understand that Valgrind gdbserver watchpoints have no length limit.

Memcheck implements hardware watchpoint simulation by marking the watched address ranges as being unaddressable. When a hardware watchpoint is removed, the range is marked as addressable and defined. Hardware watchpoint simulation of addressable-but-undefined memory zones works properly, but has the undesirable side effect of marking the zone as defined when the watchpoint is removed.

Write watchpoints might not be reported at the exact instruction that writes the monitored area, unless option `--vgdb=full` is given. Read watchpoints will always be reported at the exact instruction reading the watched memory.

It is better to avoid using hardware watchpoint of not addressable (yet) memory: in such a case, GDB will fall back to extremely slow software watchpoints. Also, if you do not quit GDB between two debugging sessions, the hardware watchpoints of the previous sessions will be re-inserted as software watchpoints if the watched memory zone is not addressable at program startup.

- Stepping inside shared libraries on ARM.

  For unknown reasons, stepping inside shared libraries on ARM may fail. A workaround is to use the `ldd` command to find the list of shared libraries and their loading address and inform GDB of the loading address using the GDB command "add-symbol-file". Example:

```
(gdb) shell ldd ./prog
 libc.so.6 => /lib/libc.so.6 (0x4002c000)
 /lib/ld-linux.so.3 (0x40000000)
(gdb) add-symbol-file /lib/libc.so.6 0x4002c000
add symbol table from file "/lib/libc.so.6" at
 .text_addr = 0x4002c000
(y or n) y
Reading symbols from /lib/libc.so.6...(no debugging symbols found)...done.
(gdb)
```

- GDB version needed for ARM and PPC32/64.

  You must use a GDB version which is able to read XML target description sent by a gdbserver. This is the standard setup if GDB was configured and built with the "expat" library. If your GDB was not configured with XML support, it will report an error message when using the "target" command. Debugging will not work because GDB will then not be able to fetch the registers from the Valgrind gdbserver. For ARM programs using the Thumb instruction set, you must use a GDB version of 7.1 or later, as earlier versions have problems with next/step/breakpoints in Thumb code.

- Stack unwinding on PPC32/PPC64.

  On PPC32/PPC64, stack unwinding for leaf functions (functions that do not call any other functions) works properly only when you give the option `--vex-iropt-register-updates=allregs-at-mem-access` or `--vex-iropt-register-updates=allregs-at-each-insn`. You must also pass this option in order to get a precise stack when a signal is trapped by GDB.

- Breakpoints encountered multiple times.

  Some instructions (e.g. x86 "rep movsb") are translated by Valgrind using a loop. If a breakpoint is placed on such an instruction, the breakpoint will be encountered multiple times -- once for each step of the "implicit" loop implementing the instruction.

- Execution of Inferior function calls by the Valgrind gdbserver.

  GDB allows the user to "call" functions inside the process being debugged. Such calls are named "inferior calls" in the GDB terminology. A typical use of an inferior call is to execute a function that prints a human-readable version of a complex data structure. To make an inferior call, use the GDB "print" command followed by the function to call and its arguments. As an example, the following GDB command causes an inferior call to the libc "printf" function to be executed by the process being debugged:

  ```
  (gdb) p printf("process being debugged has pid %d\n", getpid())
  $5 = 36
  (gdb)
  ```

  The Valgrind gdbserver supports inferior function calls. Whilst an inferior call is running, the Valgrind tool will report errors as usual. If you do not want to have such errors stop the execution of the inferior call, you can use `v.set vgdb-error` to set a big value before the call, then manually reset it to its original value when the call is complete.

  To execute inferior calls, GDB changes registers such as the program counter, and then continues the execution of the program. In a multithreaded program, all threads are continued, not just the thread instructed to make the inferior call. If another thread reports an error or encounters a breakpoint, the evaluation of the inferior call is abandoned.

  Note that inferior function calls are a powerful GDB feature, but should be used with caution. For example, if the program being debugged is stopped inside the function "printf", forcing a recursive call to printf via an inferior call will very probably create problems. The Valgrind tool might also add another level of complexity to inferior calls, e.g. by reporting tool errors during the Inferior call or due to the instrumentation done.

- Connecting to or interrupting a Valgrind process blocked in a system call.

  Connecting to or interrupting a Valgrind process blocked in a system call requires the "ptrace" system call to be usable. This may be disabled in your kernel for security reasons.

  When running your program, Valgrind's scheduler periodically checks whether there is any work to be handled by the gdbserver. Unfortunately this check is only done if at least one thread of the process is runnable. If all the threads of the process are blocked in a system call, then the checks do not happen, and the Valgrind scheduler will not invoke the gdbserver. In such a case, the vgdb relay application will "force" the gdbserver to be invoked, without the intervention of the Valgrind scheduler.

  Such forced invocation of the Valgrind gdbserver is implemented by vgdb using ptrace system calls. On a properly implemented kernel, the ptrace calls done by vgdb will not influence the behaviour of the program running under Valgrind. If however they do, giving the option `--max-invoke-ms=0` to the vgdb relay application will disable the usage of ptrace calls. The consequence of disabling ptrace usage in vgdb is that a Valgrind process blocked in a system call cannot be woken up or interrupted from GDB until it executes enough basic blocks to let the Valgrind scheduler's normal checking take effect.

  When ptrace is disabled in vgdb, you can increase the responsiveness of the Valgrind gdbserver to commands or interrupts by giving a lower value to the option `--vgdb-poll`. If your application is blocked in system calls most of the time, using a very low value for `--vgdb-poll` will cause a the gdbserver to be invoked sooner. The gdbserver polling done by Valgrind's scheduler is very efficient, so the increased polling frequency should not cause significant performance degradation.

When ptrace is disabled in vgdb, a query packet sent by GDB may take significant time to be handled by the Valgrind gdbserver. In such cases, GDB might encounter a protocol timeout. To avoid this, you can increase the value of the timeout by using the GDB command "set remotetimeout".

Ubuntu versions 10.10 and later may restrict the scope of ptrace to the children of the process calling ptrace. As the Valgrind process is not a child of vgdb, such restricted scoping causes the ptrace calls to fail. To avoid that, Valgrind will automatically allow all processes belonging to the same userid to "ptrace" a Valgrind process, by using PR_SET_PTRACER.

Unblocking processes blocked in system calls is not currently implemented on Mac OS X and Android. So you cannot connect to or interrupt a process blocked in a system call on Mac OS X or Android.

Unblocking processes blocked in system calls is implemented via agent thread on Solaris. This is quite a different approach than using ptrace on Linux, but leads to equivalent result - Valgrind gdbserver is invoked. Note that agent thread is a Solaris OS feature and cannot be disabled.

- Changing register values.

  The Valgrind gdbserver will only modify the values of the thread's registers when the thread is in status Runnable or Yielding. In other states (typically, WaitSys), attempts to change register values will fail. Amongst other things, this means that inferior calls are not executed for a thread which is in a system call, since the Valgrind gdbserver does not implement system call restart.

- Unsupported GDB functionality.

  GDB provides a lot of debugging functionality and not all of it is supported. Specifically, the following are not supported: reversible debugging and tracepoints.

- Unknown limitations or problems.

  The combination of GDB, Valgrind and the Valgrind gdbserver probably has unknown other limitations and problems. If you encounter strange or unexpected behaviour, feel free to report a bug. But first please verify that the limitation or problem is not inherent to GDB or the GDB remote protocol. You may be able to do so by checking the behaviour when using standard gdbserver part of the GDB package.

3.2.10. vgdb command line options

Usage: `vgdb [OPTION]... [[-c] COMMAND]...`

vgdb ("Valgrind to GDB") is a small program that is used as an intermediary between Valgrind and GDB or a shell. It has three usage modes:

1. As a standalone utility, it is used from a shell command line to send monitor commands to a process running under Valgrind. For this usage, the vgdb OPTION(s) must be followed by the monitor command to send. To send more than one command, separate them with the `-c` option.

2. In combination with GDB "target remote |" command, it is used as the relay application between GDB and the Valgrind gdbserver. For this usage, only OPTION(s) can be given, but no COMMAND can be given.

3. In the `--multi` mode, vgdb uses the extended remote protocol to communicate with GDB. This allows you to view output from both valgrind and GDB in the GDB session. This is accomplished via the "target extended-remote | vgdb --multi". In this mode you no longer need to start valgrind yourself. vgdb will start up valgrind when gdb tells it to run a new program. For this usage, the vgdb OPTIONS(s) can also include `--valgrind` and `--vargs` to describe how valgrind should be started.

`vgdb` accepts the following options:

`--pid=<number>`

Specifies the PID of the process to which vgdb must connect to. This option is useful in case more than one Valgrind gdbserver can be connected to. If the `--pid` argument is not given and multiple Valgrind gdbserver processes are running, vgdb will report the list of such processes and then exit.

`--vgdb-prefix`

Must be given to both Valgrind and vgdb if you want to change the default prefix for the FIFOs (named pipes) used for communication between the Valgrind gdbserver and vgdb.

`--wait=<number>`

Instructs vgdb to search for available Valgrind gdbservers for the specified number of seconds. This makes it possible start a vgdb process before starting the Valgrind gdbserver with which you intend the vgdb to communicate. This option is useful when used in conjunction with a `--vgdb-prefix` that is unique to the process you want to wait for. Also, if you use the `--wait` argument in the GDB "target remote" command, you must set the GDB remotetimeout to a value bigger than the --wait argument value. See option `--max-invoke-ms` (just below) for an example of setting the remotetimeout value.

`--max-invoke-ms=<number>`

Gives the number of milliseconds after which vgdb will force the invocation of gdbserver embedded in Valgrind. The default value is 100 milliseconds. A value of 0 disables forced invocation. The forced invocation is used when vgdb is connected to a Valgrind gdbserver, and the Valgrind process has all its threads blocked in a system call.

If you specify a large value, you might need to increase the GDB "remotetimeout" value from its default value of 2 seconds. You should ensure that the timeout (in seconds) is bigger than the `--max-invoke-ms` value. For example, for `--max-invoke-ms=5000`, the following GDB command is suitable:

```
    (gdb) set remotetimeout 6
```

`--cmd-time-out=<number>`

Instructs a standalone vgdb to exit if the Valgrind gdbserver it is connected to does not process a command in the specified number of seconds. The default value is to never time out.

`--port=<portnr>`

Instructs vgdb to use tcp/ip and listen for GDB on the specified port nr rather than to use a pipe to communicate with GDB. Using tcp/ip allows to have GDB running on one computer and debugging a Valgrind process running on another target computer. Example:

```
# On the target computer, start your program under valgrind using
valgrind --vgdb-error=0 prog
# and then in another shell, run:
vgdb --port=1234
```

On the computer which hosts GDB, execute the command:

```
gdb prog
(gdb) target remote targetip:1234
```

where targetip is the ip address or hostname of the target computer.

`--vgdb-multi`

Makes vgdb start in extended-remote mode and to wait for gdb to tell us what to run.

`--valgrind`

The path to valgrind to use, in extended-remote mode. If not specified, the system valgrind will be launched.

`--vargs`

Options to run valgrind with, in extended-remote mode. For example `-q`. Everything following `--vargs` will be provided as arguments to valgrind as is.

`-c`

To give more than one command to a standalone vgdb, separate the commands by an option `-c`. Example:

```
vgdb v.set log_output -c leak_check any
```

`-l`

Instructs a standalone vgdb to report the list of the Valgrind gdbserver processes running and then exit.

`-T`

Instructs vgdb to add timestamps to vgdb information messages.

`-D`

Instructs a standalone vgdb to show the state of the shared memory used by the Valgrind gdbserver. vgdb will exit after having shown the Valgrind gdbserver shared memory state.

`-d`

Instructs vgdb to produce debugging output. Give multiple `-d` args to increase the verbosity. When giving `-d` to a relay vgdb, you better redirect the standard error (stderr) of vgdb to a file to avoid interaction between GDB and vgdb debugging output.

3.2.11. Valgrind monitor commands

This section describes the Valgrind monitor commands, available regardless of the Valgrind tool selected. For the tool specific commands, refer to Memcheck Monitor Commands, Helgrind Monitor Commands, Callgrind Monitor Commands and Massif Monitor Commands.

The monitor commands can be sent either from a shell command line, by using a standalone vgdb, or from GDB, by using GDB's "monitor" command (see Monitor command handling by the Valgrind gdbserver) or by GDB's "valgrind" front end commands (see GDB front end commands for Valgrind gdbserver monitor commands). They can also be launched by the client program, using the VALGRIND_MONITOR_COMMAND client request.

Whatever the way the monitor command is launched, it will behave the same way. However, using the GDB's valgrind front end commands allows to benefit from the GDB infrastructure, such as expression evaluation. When relevant, the description of a monitor command below describes the additional flexibility provided by the GDB valgrind front end command. To launch a valgrind monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `valgrind` command (or the shorter aliases `vg` or `v`). In GDB, you can use `help valgrind` to get help about the valgrind front end monitor commands and you can use `apropos valgrind` to get all the commands mentionning the word "valgrind" in their name or on-line help.

- `help [debug]` instructs Valgrind's gdbserver to give the list of all monitor commands of the Valgrind core and of the tool. The optional "debug" argument tells to also give help for the monitor commands aimed at Valgrind internals debugging.

  Note that this monitor command produces the help information as provided by valgrind gdbserver. The GDB help given e.g. by `help valgrind` or `help valgrind v.info` provides the help of the GDB front end command for the equivalent valgrind gdbserver monitor command. This "GDB help" describes the additional flexibility provided by the GDB front end command.

- `v.info all_errors [also_suppressed]` shows all errors found so far.

  The optional "also_suppressed" argument indicates to also output the suppressed errors.

- `v.info last_error` shows the last error found.

- `v.info location <addr>` outputs information about the location <addr>. Possibly, the following are described: global variables, local (stack) variables, allocated or freed blocks, ... The information produced depends on the tool and on the options given to valgrind. Some tools (e.g. memcheck and helgrind) produce more detailed information for client heap blocks. For example, these tools show the stacktrace where the heap block was allocated. If a tool does not replace the malloc/free/... functions, then client heap blocks will not be described. Use the option `--read-var-info=yes` to obtain more detailed information about global or local (stack) variables.

```
(gdb) monitor v.info location 0x1130a0
 Location 0x1130a0 is 0 bytes inside global var "mx"
 declared at tc19_shadowmem.c:19
(gdb) mo v.in loc 0x1ffefffe10
 Location 0x1ffefffe10 is 0 bytes inside info.child,
 declared at tc19_shadowmem.c:139, in frame #1 of thread 1
(gdb)
```

  The GDB valgrind front end command `valgrind v.info location ADDR` accepts any address expression for its ADDR argument. In the below examples, mx is a global struct and info is a pointer to a structure. Instead of having to print the addresses of the structure and printing the pointer variable, you can directly use the expressions in the GDB valgrind front end command argument.

```
(gdb) valgrind v.info location &mx
 Location 0x1130a0 is 0 bytes inside global var "mx"
 declared at tc19_shadowmem.c:19
(gdb) v v.i lo info
 Location 0x1ffefffe10 is 0 bytes inside info.child,
 declared at tc19_shadowmem.c:139, in frame #1 of thread 1
(gdb)
```

- `v.info n_errs_found [msg]` shows the number of errors found so far, the nr of errors shown so far and the current value of the `--vgdb-error` argument. The optional `msg` (one or more words) is appended. Typically, this can be used to insert markers in a process output file between several tests executed in sequence by a process started only once. This allows to associate the errors reported by Valgrind with the specific test that produced these errors.

- `v.info open_fds` shows the list of open file descriptors and details related to the file descriptor. This only works if `--track-fds=yes` or `--track-fds=all` (to include `stdin`, `stdout` and `stderr`) was given at Valgrind startup.

- v.clo <clo_option>... changes one or more dynamic command line options. If no clo_option is given, lists the dynamically changeable options. See Dynamically Change Options. The below shows example of changing the value of --vgdb-error using directly the valgrind monitor command, using the equivalent GDB valgrind front end command. It also shows how a more flexible setting can be done using the GDB eval command.

```
(gdb) mo v.clo --vgdb-error=10
==2808839== Handling new value --vgdb-error=10 for option --vgdb-error
(gdb) v v.clo --vgdb-error=11
==2808839== Handling new value --vgdb-error=11 for option --vgdb-error
(gdb) set var $nnn = 15
(gdb) eval "v v.clo --vgdb-error=%d", $nnn + 1
==2808839== Handling new value --vgdb-error=16 for option --vgdb-error
(gdb)
```

- v.set {gdb_output | log_output | mixed_output} allows redirection of the Valgrind output (e.g. the errors detected by the tool). The default setting is mixed_output.

  With mixed_output, the Valgrind output goes to the Valgrind log (typically stderr) while the output of the interactive GDB monitor commands (e.g. v.info last_error) is displayed by GDB.

  With gdb_output, both the Valgrind output and the interactive GDB monitor commands output are displayed by GDB.

  With log_output, both the Valgrind output and the interactive GDB monitor commands output go to the Valgrind log.

- v.wait [ms (default 0)] instructs Valgrind gdbserver to sleep "ms" milli-seconds and then continue. When sent from a standalone vgdb, if this is the last command, the Valgrind process will continue the execution of the guest process. The typical usage of this is to use vgdb to send a "no-op" command to a Valgrind gdbserver so as to continue the execution of the guest process.

  The GDB valgrind front end command valgrind v.wait MS accepts any integer expression for its MS argument, while the monitor command accepts only integer numbers.

- v.kill requests the gdbserver to kill the process. This can be used from a standalone vgdb to properly kill a Valgrind process which is currently expecting a vgdb connection.

- v.set vgdb-error <errornr> dynamically changes the value of the --vgdb-error valgrind command line argument. A typical usage of this is to start with --vgdb-error=0 on the command line, then set a few breakpoints, set the vgdb-error value to a huge value and continue execution. Note that you can also change this value using e.g. v.clo --vgdb-error=12

  The GDB valgrind front end command valgrind v.set vgdb-error NUM accepts any integer expression for its ERRORNR argument, while the monitor command accepts only integer numbers.

- v.set merge-recursive-frames <num> dynamically changes the value of the --merge-recursive-frames valgrind command line argument. Note that you can also change this value using e.g. v.clo --merge-recursive-frames=5

  The GDB valgrind front end command valgrind v.set merge-recursive-frames NUM accepts any integer expression for its NUM argument, while the monitor command accepts only integer numbers.

- xtmemory [<filename> default xtmemory.kcg.%p.%n] requests the tool (Memcheck, Massif, Helgrind) to produce an xtree heap memory report. See Execution Trees for a detailed explanation about execution trees.

The following Valgrind monitor commands are useful for investigating the behaviour of Valgrind or its gdbserver in case of problems or bugs.

- v.do expensive_sanity_check_general executes various sanity checks. In particular, the sanity of the Valgrind heap is verified. This can be useful if you suspect that your program and/or Valgrind has a bug corrupting Valgrind data structure. It can also be used when a Valgrind tool reports a client error to the connected GDB, in order to verify the sanity of Valgrind before continuing the execution.

- v.info gdbserver_status shows the gdbserver status. In case of problems (e.g. of communications), this shows the values of some relevant Valgrind gdbserver internal variables. Note that the variables related to breakpoints and watchpoints (e.g. the number of breakpoint addresses and the number of watchpoints) will be zero, as GDB by default removes all watchpoints and breakpoints when execution stops, and re-inserts them when resuming the execution of the debugged process. You can change this GDB behaviour by using the GDB command set breakpoint always-inserted on.

- v.info memory [aspacemgr] shows the statistics of Valgrind's internal heap management. If option --profile-heap=yes was given, detailed statistics will be output. With the optional argument aspacemgr. the segment list maintained by valgrind address space manager will be output. Note that this list of segments is always output on the Valgrind log.

- v.info exectxt shows information about the "executable contexts" (i.e. the stack traces) recorded by Valgrind. For some programs, Valgrind can record a very high number of such stack traces, causing a high memory usage. This monitor command shows all the recorded stack traces, followed by some statistics. This can be used to analyse the reason for having a big number of stack traces. Typically, you will use this command if v.info memory has shown significant memory usage by the "exectxt" arena.

- v.info scheduler shows various information about threads. First, it outputs the host stack trace, i.e. the Valgrind code being executed. Then, for each thread, it outputs the thread state. For non terminated threads, the state is followed by the guest (client) stack trace. Finally, for each active thread or for each terminated thread slot not yet re-used, it shows the max usage of the valgrind stack.

  Showing the client stack traces allows to compare the stack traces produced by the Valgrind unwinder with the stack traces produced by GDB+Valgrind gdbserver. Pay attention that GDB and Valgrind scheduler status have their own thread numbering scheme. To make the link between the GDB thread number and the corresponding Valgrind scheduler thread number, use the GDB command info threads. The output of this command shows the GDB thread number and the valgrind 'tid'. The 'tid' is the thread number output by v.info scheduler. When using the callgrind tool, the callgrind monitor command status outputs internal callgrind information about the stack/call graph it maintains.

- v.info stats shows various valgrind core and tool statistics. With this, Valgrind and tool statistics can be examined while running, even without option --stats=yes.

- v.info unwind <addr> [<len>] shows the CFI unwind debug info for the address range [addr, addr+len-1]. The default value of <len> is 1, giving the unwind information for the instruction at <addr>.

  The GDB valgrind front end command valgrind v.info unwind ADDR [LEN] accepts any address expression for its first ADDR argument, such as $pc. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

- v.set debuglog <intvalue> sets the Valgrind debug log level to <intvalue>. This allows to dynamically change the log level of Valgrind e.g. when a problem is detected.

  The GDB valgrind front end command valgrind v.set debuglog LEVEL accepts any address expression for its LEVEL argument.

- v.set hostvisibility [yes*|no] The value "yes" indicates to gdbserver that GDB can look at the Valgrind 'host' (internal) status/memory. "no" disables this access. When hostvisibility is activated, GDB can e.g.

look at Valgrind global variables. As an example, to examine a Valgrind global variable of the memcheck tool on an x86, do the following setup:

```
(gdb) monitor v.set hostvisibility yes
Enabled access to Valgrind memory/status by GDB
If not yet done, tell GDB which valgrind file(s) to use, typically:
add-symbol-file /home/philippe/valgrind/git/improve/Inst/libexec/valgrind/memcheck-amd64-
(gdb) add-symbol-file /home/philippe/valgrind/git/improve/Inst/libexec/valgrind/memcheck-
add symbol table from file "/home/philippe/valgrind/git/improve/Inst/libexec/valgrind/mem
 .text_addr = 0x58001000
(y or n) y
Reading symbols from /home/philippe/valgrind/git/improve/Inst/libexec/valgrind/memcheck-a
(gdb)
```

After that, variables defined in memcheck-x86-linux can be accessed, e.g.

```
(gdb) p /x vgPlain_threads[1].os_state
$3 = {lwpid = 0x4688, threadgroup = 0x4688, parent = 0x0,
  valgrind_stack_base = 0x62e78000, valgrind_stack_init_SP = 0x62f79fe0,
  exitcode = 0x0, fatalsig = 0x0}
(gdb) p vex_control
$5 = {iropt_verbosity = 0, iropt_level = 2,
  iropt_register_updates = VexRegUpdUnwindregsAtMemAccess,
  iropt_unroll_thresh = 120, guest_max_insns = 60, guest_chase_thresh = 10}
(gdb)
```

- `v.translate <address> [<traceflags>]` shows the translation of the block containing `address` with the given trace flags. The `traceflags` value bit patterns have similar meaning to Valgrind's `--trace-flags` option. It can be given in hexadecimal (e.g. 0x20) or decimal (e.g. 32) or in binary 1s and 0s bit (e.g. 0b00100000). The default value of the traceflags is 0b00100000, corresponding to "show after instrumentation". The output of this command always goes to the Valgrind log.

  The additional bit flag 0b100000000 (bit 8) has no equivalent in the `--trace-flags` option. It enables tracing of the gdbserver specific instrumentation. Note that this bit 8 can only enable the addition of gdbserver instrumentation in the trace. Setting it to 0 will not disable the tracing of the gdbserver instrumentation if it is active for some other reason, for example because there is a breakpoint at this address or because gdbserver is in single stepping mode.

  The GDB valgrind front end command `valgrind v.translate ADDR [TRACEFLAG]` accepts any address expression for its first ADDR argument, such as $pc. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

3.3. Function wrapping

Valgrind allows calls to some specified functions to be intercepted and rerouted to a different, user-supplied function. This can do whatever it likes, typically examining the arguments, calling onwards to the original, and possibly examining the result. Any number of functions may be wrapped.

Function wrapping is useful for instrumenting an API in some way. For example, Helgrind wraps functions in the POSIX pthreads API so it can know about thread status changes, and the core is able to wrap functions in the MPI (message-passing) API so it can know of memory status changes associated with message arrival/departure. Such information is usually passed to Valgrind by using client requests in the wrapper functions, although the exact mechanism may vary.

3.3.1. A Simple Example

Supposing we want to wrap some function

```
int foo ( int x, int y ) { return x + y; }
```

A wrapper is a function of identical type, but with a special name which identifies it as the wrapper for `foo`. Wrappers need to include supporting macros from `valgrind.h`. Here is a simple wrapper which prints the arguments and return value:

```
#include <stdio.h>
#include "valgrind.h"
int I_WRAP_SONAME_FNNAME_ZU(NONE,foo)( int x, int y )
{
   int    result;
   OrigFn fn;
   VALGRIND_GET_ORIG_FN(fn);
   printf("foo's wrapper: args %d %d\n", x, y);
   CALL_FN_W_WW(result, fn, x,y);
   printf("foo's wrapper: result %d\n", result);
   return result;
}
```

To become active, the wrapper merely needs to be present in a text section somewhere in the same process' address space as the function it wraps, and for its ELF symbol name to be visible to Valgrind. In practice, this means either compiling to a `.o` and linking it in, or compiling to a `.so` and `LD_PRELOAD`ing it in. The latter is more convenient in that it doesn't require relinking.

All wrappers have approximately the above form. There are three crucial macros:

`I_WRAP_SONAME_FNNAME_ZU`: this generates the real name of the wrapper. This is an encoded name which Valgrind notices when reading symbol table information. What it says is: I am the wrapper for any function named `foo` which is found in an ELF shared object with an empty ("`NONE`") soname field. The specification mechanism is powerful in that wildcards are allowed for both sonames and function names. The details are discussed below.

`VALGRIND_GET_ORIG_FN`: once in the wrapper, the first priority is to get hold of the address of the original (and any other supporting information needed). This is stored in a value of opaque type `OrigFn`. The information is acquired using `VALGRIND_GET_ORIG_FN`. It is crucial to make this macro call before calling any other wrapped function in the same thread.

`CALL_FN_W_WW`: eventually we will want to call the function being wrapped. Calling it directly does not work, since that just gets us back to the wrapper and leads to an infinite loop. Instead, the result lvalue, `OrigFn` and arguments are handed to one of a family of macros of the form `CALL_FN_*`. These cause Valgrind to call the original and avoid recursion back to the wrapper.

3.3.2. Wrapping Specifications

This scheme has the advantage of being self-contained. A library of wrappers can be compiled to object code in the normal way, and does not rely on an external script telling Valgrind which wrappers pertain to which originals.

Each wrapper has a name which, in the most general case says: I am the wrapper for any function whose name matches FNPATT and whose ELF "soname" matches SOPATT. Both FNPATT and SOPATT may contain wildcards (asterisks) and other characters (spaces, dots, @, etc) which are not generally regarded as valid C identifier names.

This flexibility is needed to write robust wrappers for POSIX pthread functions, where typically we are not completely sure of either the function name or the soname, or alternatively we want to wrap a whole set of functions at once.

For example, `pthread_create` in GNU libpthread is usually a versioned symbol - one whose name ends in, eg, `@GLIBC_2.3`. Hence we are not sure what its real name is. We also want to cover any soname of the form `libpthread.so*`. So the header of the wrapper will be

```
int I_WRAP_SONAME_FNNAME_ZZ(libpthreadZdsoZd0,pthreadZucreateZAZa)
  ( ... formals ... )
  { ... body ... }
```

In order to write unusual characters as valid C function names, a Z-encoding scheme is used. Names are written literally, except that a capital Z acts as an escape character, with the following encoding:

```
     Za   encodes    *
     Zp              +
     Zc              :
     Zd              .
     Zu              _
     Zh              -
     Zs              (space)
     ZA              @
     ZZ              Z
     ZL              (       # only in valgrind 3.3.0 and later
     ZR              )       # only in valgrind 3.3.0 and later
```

Hence `libpthreadZdsoZd0` is an encoding of the soname `libpthread.so.0` and `pthreadZucreateZAZa` is an encoding of the function name `pthread_create@*`.

The macro `I_WRAP_SONAME_FNNAME_ZZ` constructs a wrapper name in which both the soname (first component) and function name (second component) are Z-encoded. Encoding the function name can be tiresome and is often unnecessary, so a second macro, `I_WRAP_SONAME_FNNAME_ZU`, can be used instead. The `_ZU` variant is also useful for writing wrappers for C++ functions, in which the function name is usually already mangled using some other convention in which Z plays an important role. Having to encode a second time quickly becomes confusing.

Since the function name field may contain wildcards, it can be anything, including just `*`. The same is true for the soname. However, some ELF objects - specifically, main executables - do not have sonames. Any object lacking a soname is treated as if its soname was `NONE`, which is why the original example above had a name `I_WRAP_SONAME_FNNAME_ZU(NONE,foo)`.

Note that the soname of an ELF object is not the same as its file name, although it is often similar. You can find the soname of an object `libfoo.so` using the command `readelf -a libfoo.so | grep soname`.

3.3.3. Wrapping Semantics

The ability for a wrapper to replace an infinite family of functions is powerful but brings complications in situations where ELF objects appear and disappear (are dlopen'd and dlclose'd) on the fly. Valgrind tries to maintain sensible behaviour in such situations.

For example, suppose a process has dlopened (an ELF object with soname) `object1.so`, which contains `function1`. It starts to use `function1` immediately.

After a while it dlopens `wrappers.so`, which contains a wrapper for `function1` in (soname) `object1.so`. All subsequent calls to `function1` are rerouted to the wrapper.

If `wrappers.so` is later dlclose'd, calls to `function1` are naturally routed back to the original.

Alternatively, if `object1.so` is dlclose'd but `wrappers.so` remains, then the wrapper exported by `wrappers.so` becomes inactive, since there is no way to get to it - there is no original to call any more. However, Valgrind remembers that the wrapper is still present. If `object1.so` is eventually dlopen'd again, the wrapper will become active again.

In short, valgrind inspects all code loading/unloading events to ensure that the set of currently active wrappers remains consistent.

A second possible problem is that of conflicting wrappers. It is easily possible to load two or more wrappers, both of which claim to be wrappers for some third function. In such cases Valgrind will complain about conflicting wrappers when the second one appears, and will honour only the first one.

3.3.4. Debugging

Figuring out what's going on given the dynamic nature of wrapping can be difficult. The `--trace-redir=yes` option makes this possible by showing the complete state of the redirection subsystem after every `mmap`/`munmap` event affecting code (text).

There are two central concepts:

- A "redirection specification" is a binding of a (soname pattern, fnname pattern) pair to a code address. These bindings are created by writing functions with names made with the `I_WRAP_SONAME_FNNAME_{ZZ,_ZU}` macros.
- An "active redirection" is a code-address to code-address binding currently in effect.

The state of the wrapping-and-redirection subsystem comprises a set of specifications and a set of active bindings. The specifications are acquired/discarded by watching all `mmap`/`munmap` events on code (text) sections. The active binding set is (conceptually) recomputed from the specifications, and all known symbol names, following any change to the specification set.

`--trace-redir=yes` shows the contents of both sets following any such event.

`-v` prints a line of text each time an active specification is used for the first time.

Hence for maximum debugging effectiveness you will need to use both options.

One final comment. The function-wrapping facility is closely tied to Valgrind's ability to replace (redirect) specified functions, for example to redirect calls to `malloc` to its own implementation. Indeed, a replacement function can be regarded as a wrapper function which does not call the original. However, to make the implementation more robust, the two kinds of interception (wrapping vs replacement) are treated differently.

`--trace-redir=yes` shows specifications and bindings for both replacement and wrapper functions. To differentiate the two, replacement bindings are printed using `R->` whereas wraps are printed using `W->`.

3.3.5. Limitations - control flow

For the most part, the function wrapping implementation is robust. The only important caveat is: in a wrapper, get hold of the `OrigFn` information using `VALGRIND_GET_ORIG_FN` before calling any other wrapped function. Once you have the `OrigFn`, arbitrary calls between, recursion between, and longjumps out of wrappers should work correctly. There is never any interaction between wrapped functions and merely replaced functions (eg `malloc`), so you can call `malloc` etc safely from within wrappers.

The above comments are true for {x86,amd64,ppc32,arm,mips32,s390}-linux. On ppc64-linux function wrapping is more fragile due to the (arguably poorly designed) ppc64-linux ABI. This mandates the use of a shadow stack which tracks entries/exits of both wrapper and replacement functions. This gives two limitations: firstly, longjumping out of wrappers will rapidly lead to disaster, since the shadow stack will not get correctly cleared. Secondly, since the shadow stack has finite size, recursion between wrapper/replacement functions is only possible to a limited depth, beyond which Valgrind has to abort the run. This depth is currently 16 calls.

For all platforms ({x86,amd64,ppc32,ppc64,arm,mips32,s390}-linux) all the above comments apply on a per-thread basis. In other words, wrapping is thread-safe: each thread must individually observe the above restrictions, but there is no need for any kind of inter-thread cooperation.

3.3.6. Limitations - original function signatures

As shown in the above example, to call the original you must use a macro of the form `CALL_FN_*`. For technical reasons it is impossible to create a single macro to deal with all argument types and numbers, so a family of macros covering the most common cases is supplied. In what follows, 'W' denotes a machine-word-typed value (a pointer or a C `long`), and 'v' denotes C's `void` type. The currently available macros are:

```
CALL_FN_v_v    -- call an original of type  void fn ( void )
CALL_FN_W_v    -- call an original of type  long fn ( void )

CALL_FN_v_W    -- call an original of type  void fn ( long )
CALL_FN_W_W    -- call an original of type  long fn ( long )

CALL_FN_v_WW   -- call an original of type  void fn ( long, long )
CALL_FN_W_WW   -- call an original of type  long fn ( long, long )

CALL_FN_v_WWW  -- call an original of type  void fn ( long, long, long )
CALL_FN_W_WWW  -- call an original of type  long fn ( long, long, long )

CALL_FN_W_WWWW -- call an original of type  long fn ( long, long, long, long )
CALL_FN_W_5W   -- call an original of type  long fn ( long, long, long, long, long )
CALL_FN_W_6W   -- call an original of type  long fn ( long, long, long, long, long, long )
and so on, up to
CALL_FN_W_12W
```

The set of supported types can be expanded as needed. It is regrettable that this limitation exists. Function wrapping has proven difficult to implement, with a certain apparently unavoidable level of ickiness. After several implementation attempts, the present arrangement appears to be the least-worst tradeoff. At least it works reliably in the presence of dynamic linking and dynamic code loading/unloading.

You should not attempt to wrap a function of one type signature with a wrapper of a different type signature. Such trickery will surely lead to crashes or strange behaviour. This is not a limitation of the function wrapping implementation, merely a reflection of the fact that it gives you sweeping powers to shoot yourself in the foot if you are not careful. Imagine the instant havoc you could wreak by writing a wrapper which matched any function name in any soname - in effect, one which claimed to be a wrapper for all functions in the process.

3.3.7. Examples

In the source tree, `memcheck/tests/wrap[1-8].c` provide a series of examples, ranging from very simple to quite advanced.

`mpi/libmpiwrap.c` is an example of wrapping a big, complex API (the MPI-2 interface). This file defines almost 300 different wrappers.

4. Memcheck: a memory error detector

To use this tool, you may specify --tool=memcheck on the Valgrind command line. You don't have to, though, since Memcheck is the default tool.

4.1. Overview

Memcheck is a memory error detector. It can detect the following problems that are common in C and C++ programs.

- Accessing memory you shouldn't, e.g. overrunning and underrunning heap blocks, overrunning the top of the stack, and accessing memory after it has been freed.
- Using undefined values, i.e. values that have not been initialised, or that have been derived from other undefined values.
- Incorrect freeing of heap memory, such as double-freeing heap blocks, or mismatched use of malloc/new/new[] versus free/delete/delete[]

  Mismatches will also be reported for sized and aligned allocation and deallocation functions if the deallocation value does not match the allocation value.
- Overlapping src and dst pointers in memcpy and related functions.
- Passing a fishy (presumably negative) value to the size parameter of a memory allocation function.
- Using a size value of 0 with realloc.
- Using an alignment value that is not a power of two.
- Memory leaks.

Problems like these can be difficult to find by other means, often remaining undetected for long periods, then causing occasional, difficult-to-diagnose crashes.

Memcheck also provides Execution Trees memory profiling using the command line option --xtree-memory and the monitor command xtmemory.

4.2. Explanation of error messages from Memcheck

Memcheck issues a range of error messages. This section presents a quick summary of what error messages mean. The precise behaviour of the error-checking machinery is described in Details of Memcheck's checking machinery.

4.2.1. Illegal read / Illegal write errors

For example:

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40B07FF4: read_png_image(QImageIO *) (kernel/qpngio.cpp:326)
   by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
```

```
Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn't. In this example, the program did a 4-byte read at address 0xBFFFF0E0, somewhere within the system-supplied library libpng.so.2.1.0.9, which was called from somewhere else in the same library, called from line 326 of `qpngio.cpp`, and so on.

Memcheck tries to establish what the illegal address might relate to, since that's often useful. So, if it points into a block of memory which has already been freed, you'll be informed of this, and also where the block was freed. Likewise, if it should turn out to be just off the end of a heap block, a common result of off-by-one-errors in array subscripting, you'll be informed of this fact, and also where the block was allocated. If you use the `--read-var-info` option Memcheck will run more slowly but may give a more detailed description of any illegal address.

In this example, Memcheck can't identify the address. Actually the address is on the stack, but, for some reason, this is not a valid stack address -- it is below the stack pointer and that isn't allowed. In this particular case it's probably caused by GCC generating invalid code, a known bug in some ancient versions of GCC.

Note that Memcheck only tells you that your program is about to access memory at an illegal address. It can't stop the access from happening. So, if your program makes an access which normally would result in a segmentation fault, you program will still suffer the same fate -- but you will get a message from Memcheck immediately prior to this. In this particular example, reading junk on the stack is non-fatal, and the program stays alive.

4.2.2. Use of uninitialised values

For example:

```
Conditional jump or move depends on uninitialised value(s)
   at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
   by 0x402E8476: _IO_printf (printf.c:36)
   by 0x8048472: main (tests/manuel1.c:8)
```

An uninitialised-value use error is reported when your program uses a value which hasn't been initialised -- in other words, is undefined. Here, the undefined value is used somewhere inside the `printf` machinery of the C library. This error was reported when running the following small program:

```
int main()
{
  int x;
  printf ("x = %d\n", x);
}
```

It is important to understand that your program can copy around junk (uninitialised) data as much as it likes. Memcheck observes this and keeps track of the data, but does not complain. A complaint is issued only when your program attempts to make use of uninitialised data in a way that might affect your program's externally-visible behaviour. In this example, `x` is uninitialised. Memcheck observes the value being passed to `_IO_printf` and thence to `_IO_vfprintf`, but makes no comment. However, `_IO_vfprintf` has to examine the value of `x` so it can turn it into the corresponding ASCII string, and it is at this point that Memcheck complains.

Sources of uninitialised data tend to be:

- Local variables in procedures which have not been initialised, as in the example above.
- The contents of heap blocks (allocated with `malloc`, `new`, or a similar function) before you (or a constructor) write something there.

To see information on the sources of uninitialised data in your program, use the `--track-origins=yes` option. This makes Memcheck run more slowly, but can make it much easier to track down the root causes of uninitialised value errors.

4.2.3. Use of uninitialised or unaddressable values in system calls

Memcheck checks all parameters to system calls:

- It checks all the direct parameters themselves, whether they are initialised.
- Also, if a system call needs to read from a buffer provided by your program, Memcheck checks that the entire buffer is addressable and its contents are initialised.
- Also, if the system call needs to write to a user-supplied buffer, Memcheck checks that the buffer is addressable.

After the system call, Memcheck updates its tracked information to precisely reflect any changes in memory state caused by the system call.

Here's an example of two system calls with invalid parameters:

```
#include <stdlib.h>
#include <unistd.h>
int main( void )
{
  char* arr  = malloc(10);
  int*  arr2 = malloc(sizeof(int));
  write( 1 /* stdout */, arr, 10 );
  exit(arr2[0]);
}
```

You get these complaints ...

```
Syscall param write(buf) points to uninitialised byte(s)
   at 0x25A48723: __write_nocancel (in /lib/tls/libc-2.3.3.so)
   by 0x259AFAD3: __libc_start_main (in /lib/tls/libc-2.3.3.so)
   by 0x8048348: (within /auto/homes/njn25/grind/head4/a.out)
 Address 0x25AB8028 is 0 bytes inside a block of size 10 alloc'd
   at 0x259852B0: malloc (vg_replace_malloc.c:130)
   by 0x80483F1: main (a.c:5)

Syscall param exit(error_code) contains uninitialised byte(s)
   at 0x25A21B44: __GI__exit (in /lib/tls/libc-2.3.3.so)
   by 0x8048426: main (a.c:8)
```

... because the program has (a) written uninitialised junk from the heap block to the standard output, and (b) passed an uninitialised value to `exit`. Note that the first error refers to the memory pointed to by `buf` (not `buf` itself), but the second error refers directly to `exit`'s argument `arr2[0]`.

4.2.4. Illegal frees

For example:

```
Invalid free()
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
 Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
```

Memcheck keeps track of the blocks allocated by your program with `malloc`/`new`, so it can know exactly whether or not the argument to `free`/`delete` is legitimate or not. Here, this test program has freed the same block twice. As with the illegal read/write errors, Memcheck attempts to make sense of the address freed. If, as here, the address is one which has previously been freed, you wil be told that -- making duplicate frees of the same block easy to spot. You will also get this message if you try to free a pointer that doesn't point to the start of a heap block.

4.2.5. When a heap block is freed with an inappropriate deallocation function

In the following example, a block allocated with `new[]` has wrongly been deallocated with `free`:

```
Mismatched free() / delete / delete []
   at 0x40043249: free (vg_clientfuncs.c:171)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
 Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: operator new[](unsigned int) (vg_clientfuncs.c:152)
   by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
   by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
   by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

In `C++` it's important to deallocate memory in a way compatible with how it was allocated.

Most of the time in C++ you will write code that uses `new expression` and `delete expression` (see cppreference new expression and cppreference delete expression). A new expression will call `operator new` to perform the allocation and then call the constructor (if one exists) on the object. Similarly a delete expression will call the destructor on the object (if one exists) and then call `operator delete`. The array overloads call constructors/destructors for each object in the array.

The deal is:

- If allocated with `malloc`, `calloc`, `realloc`, `valloc` or `memalign`, you must deallocate with `free`.
- If allocated with `new`, you must deallocate with `delete`.
- If allocated with `new[]`, you must deallocate with `delete[]`.

Mixing types of allocators and deallocators is undefined behaviour. That means that on some platforms you might not have any problems, but the same program may then crash on a different platform, Solaris for example. So it's best to fix it properly. According to the KDE folks "it's amazing how many C++ programmers don't know this".

The reason behind the requirement is as follows. In some C++ implementations, `delete[]` must be used for objects allocated by `new[]` because the compiler stores the size of the array and the pointer-to-member to the destructor of the array's content just before the pointer actually returned. `delete` doesn't account for this and will get confused, possibly corrupting the heap. Even if there is no corruption there are likely to be resource leaks since using the wrong delete may result in the wrong number of destructors being called.

C++ aligned allocations need to be freed using aligned delete with the same alignment.

4.2.6. Overlapping source and destination blocks

The following C library functions copy some data from one memory block to another (or something similar): `memcpy`, `strcpy`, `strncpy`, `strcat`, `strncat`. The blocks pointed to by their `src` and `dst` pointers aren't allowed to overlap. The POSIX standards have wording along the lines "If copying takes place between objects that overlap, the behavior is undefined." Therefore, Memcheck checks for this.

For example:

```
==27492== Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
==27492==    at 0x40026CDC: memcpy (mc_replace_strmem.c:71)
==27492==    by 0x804865A: main (overlap.c:40)
```

You don't want the two blocks to overlap because one of them could get partially overwritten by the copying.

You might think that Memcheck is being overly pedantic reporting this in the case where `dst` is less than `src`. For example, the obvious way to implement `memcpy` is by copying from the first byte to the last. However, the optimisation guides of some architectures recommend copying from the last byte down to the first. Also, some implementations of `memcpy` zero `dst` before copying, because zeroing the destination's cache line(s) can improve performance.

The moral of the story is: if you want to write truly portable code, don't make any assumptions about the language implementation.

4.2.7. Fishy argument values

All memory allocation functions take an argument specifying the size of the memory block that should be allocated. Clearly, the requested size should be a non-negative value and is typically not excessively large. For instance, it is extremely unlikly that the size of an allocation request exceeds 2**63 bytes on a 64-bit machine. It is much more likely that such a value is the result of an erroneous size calculation and is in effect a negative value (that just happens to appear excessively large because the bit pattern is interpreted as an unsigned integer). Such a value is called a "fishy value". The `size` argument of the following allocation functions is checked for being fishy: `malloc`, `calloc`, `realloc`, `memalign`, `posix_memalign`, `aligned_alloc`, `new`, `new []`. `__builtin_new`, `__builtin_vec_new`, For `calloc` both arguments are checked.

For example:

```
==32233== Argument 'size' of function malloc has a fishy (possibly negative) value: -3
==32233==    at 0x4C2CFA7: malloc (vg_replace_malloc.c:298)
==32233==    by 0x400555: foo (fishy.c:15)
==32233==    by 0x400583: main (fishy.c:23)
```

In earlier Valgrind versions those values were being referred to as "silly arguments" and no back-trace was included.

4.2.8. Realloc size zero

The (ab)use or realloc to also do the job of `free` has been poorly understood for a long time. In the C17 standard ISO/IEC 9899:2017] the behaviour of realloc when the size argument is zero is specified as implementation defined. Memcheck warns about the non-portable use or realloc.

For example:

```
==77609== realloc() with size 0
==77609==    at 0x48502B8: realloc (vg_replace_malloc.c:1450)
==77609==    by 0x201989: main (realloczero.c:8)
==77609==  Address 0x5464040 is 0 bytes inside a block of size 4 alloc'd
==77609==    at 0x484CBB4: malloc (vg_replace_malloc.c:397)
==77609==    by 0x201978: main (realloczero.c:7)
```

4.2.9. Alignment Errors

C and C++ have several functions that allow the user to obtain aligned memory. Typically this is done for performance reasons so that the memory will be cache line or memory page aligned. C has the functions `memalign`, `posix_memalign` and `aligned_alloc`. C++ has numerous overloads of `operator new` and `operator delete`. Of these, posix_memalign is quite clearly specified, the others vary quite widely between implementations. Valgrind will generate errors for values of alignment that are invalid on any platform.

`memalign` will produce errors if the alignment is zero or not a multiple of two.

`posix_memalign` will produce errors if the alignment is less than sizeof(size_t), not a multiple of two or if the size is zero.

`aligned_alloc` will produce errors if the alignment is not a multiple of two , if the size is zero or if the size is not an integral multiple of the alignment.

`aligned new` will produce errors if the alignment is zero or not a multiple of two. The `nothrow` overloads will return a NULL pointer. The non-nothrow overloads will abort Valgrind.

`aligned delete` will produce errors if the alignment is zero or not a multiple of two or if the alignment is not the same as that used by `aligned new`.

`sized delete` will produce errors if the size is not the same as that used by `new`.

`sized aligned delete` combines the error conditions of the individual sized and aligned delete operators.

Example output:

```
==65825== Invalid alignment value: 3 (should be power of 2)
==65825==    at 0x485197E: memalign (vg_replace_malloc.c:1740)
==65825==    by 0x201CD2: main (memalign.c:39)
```

4.2.10. Memory leak detection

Memcheck keeps track of all heap blocks issued in response to calls to `malloc`/`new` et al. So when the program exits, it knows which blocks have not been freed.

If `--leak-check` is set appropriately, for each remaining block, Memcheck determines if the block is reachable from pointers within the root-set. The root-set consists of (a) general purpose registers of all threads, and (b) initialised, aligned, pointer-sized data words in accessible client memory, including stacks.

There are two ways a block can be reached. The first is with a "start-pointer", i.e. a pointer to the start of the block. The second is with an "interior-pointer", i.e. a pointer to the middle of the block. There are several ways we know of that an interior-pointer can occur:

- The pointer might have originally been a start-pointer and have been moved along deliberately (or not deliberately) by the program. In particular, this can happen if your program uses tagged pointers, i.e. if it uses the bottom one, two or three bits of a pointer, which are normally always zero due to alignment, in order to store extra information.

- It might be a random junk value in memory, entirely unrelated, just a coincidence.
- It might be a pointer to the inner char array of a C++ `std::string`. For example, some compilers add 3 words at the beginning of the std::string to store the length, the capacity and a reference count before the memory containing the array of characters. They return a pointer just after these 3 words, pointing at the char array.
- Some code might allocate a block of memory, and use the first 8 bytes to store (block size - 8) as a 64bit number. `sqlite3MemMalloc` does this.
- It might be a pointer to an array of C++ objects (which possess destructors) allocated with `new[]`. In this case, some compilers store a "magic cookie" containing the array length at the start of the allocated block, and return a pointer to just past that magic cookie, i.e. an interior-pointer. See this page for more information.
- It might be a pointer to an inner part of a C++ object using multiple inheritance.

You can optionally activate heuristics to use during the leak search to detect the interior pointers corresponding to the `stdstring`, `length64`, `newarray` and `multipleinheritance` cases. If the heuristic detects that an interior pointer corresponds to such a case, the block will be considered as reachable by the interior pointer. In other words, the interior pointer will be treated as if it were a start pointer.

With that in mind, consider the nine possible cases described by the following figure.

```
     Pointer chain            AAA Leak Case   BBB Leak Case
     -------------            -------------   -------------
(1)  RRR ------------> BBB                    DR
(2)  RRR ---> AAA ---> BBB    DR              IR
(3)  RRR               BBB                    DL
(4)  RRR      AAA ---> BBB    DL              IL
(5)  RRR ------?-----> BBB                    (y)DR, (n)DL
(6)  RRR ---> AAA -?-> BBB    DR              (y)IR, (n)DL
(7)  RRR -?-> AAA ---> BBB    (y)DR, (n)DL    (y)IR, (n)IL
(8)  RRR -?-> AAA -?-> BBB    (y)DR, (n)DL    (y,y)IR, (n,y)IL, (_,n)DL
(9)  RRR      AAA -?-> BBB    DL              (y)IL, (n)DL

Pointer chain legend:
- RRR: a root set node or DR block
- AAA, BBB: heap blocks
- --->: a start-pointer
- -?->: an interior-pointer

Leak Case legend:
- DR: Directly reachable
- IR: Indirectly reachable
- DL: Directly lost
- IL: Indirectly lost
- (y)XY: it's XY if the interior-pointer is a real pointer
- (n)XY: it's XY if the interior-pointer is not a real pointer
- (_)XY: it's XY in either case
```

Every possible case can be reduced to one of the above nine. Memcheck merges some of these cases in its output, resulting in the following four leak kinds.

- "Still reachable". This covers cases 1 and 2 (for the BBB blocks) above. A start-pointer or chain of start-pointers to the block is found. Since the block is still pointed at, the programmer could, at least in principle, have freed it before program exit. "Still reachable" blocks are very common and arguably not a problem. So, by default, Memcheck won't report such blocks individually.

- "Definitely lost". This covers case 3 (for the BBB blocks) above. This means that no pointer to the block can be found. The block is classified as "lost", because the programmer could not possibly have freed it at program exit, since no pointer to it exists. This is likely a symptom of having lost the pointer at some earlier point in the program. Such cases should be fixed by the programmer.
- "Indirectly lost". This covers cases 4 and 9 (for the BBB blocks) above. This means that the block is lost, not because there are no pointers to it, but rather because all the blocks that point to it are themselves lost. For example, if you have a binary tree and the root node is lost, all its children nodes will be indirectly lost. Because the problem will disappear if the definitely lost block that caused the indirect leak is fixed, Memcheck won't report such blocks individually by default.
- "Possibly lost". This covers cases 5--8 (for the BBB blocks) above. This means that a chain of one or more pointers to the block has been found, but at least one of the pointers is an interior-pointer. This could just be a random value in memory that happens to point into a block, and so you shouldn't consider this ok unless you know you have interior-pointers.

(Note: This mapping of the nine possible cases onto four leak kinds is not necessarily the best way that leaks could be reported; in particular, interior-pointers are treated inconsistently. It is possible the categorisation may be improved in the future.)

Furthermore, if suppressions exists for a block, it will be reported as "suppressed" no matter what which of the above four kinds it belongs to.

The following is an example leak summary.

```
LEAK SUMMARY:
   definitely lost: 48 bytes in 3 blocks.
   indirectly lost: 32 bytes in 2 blocks.
     possibly lost: 96 bytes in 6 blocks.
   still reachable: 64 bytes in 4 blocks.
        suppressed: 0 bytes in 0 blocks.
```

If heuristics have been used to consider some blocks as reachable, the leak summary details the heuristically reachable subset of 'still reachable:' per heuristic. In the below example, of the 95 bytes still reachable, 87 bytes (56+7+8+16) have been considered heuristically reachable.

```
LEAK SUMMARY:
   definitely lost: 4 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 95 bytes in 6 blocks
                      of which reachable via heuristic:
                        stdstring          : 56 bytes in 2 blocks
                        length64           : 16 bytes in 1 blocks
                        newarray           : 7 bytes in 1 blocks
                        multipleinheritance: 8 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
```

If `--leak-check=full` is specified, Memcheck will give details for each definitely lost or possibly lost block, including where it was allocated. (Actually, it merges results for all blocks that have the same leak kind and sufficiently similar stack traces into a single "loss record". The `--leak-resolution` lets you control the meaning of "sufficiently similar".) It cannot tell you when or how or why the pointer to a leaked block was lost; you have to work that out for yourself. In general, you should attempt to ensure your programs do not have any definitely lost or possibly lost blocks at exit.

For example:

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:39)

88 (8 direct, 80 indirect) bytes in 1 blocks are definitely lost in loss record 13 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:25)
```

The first message describes a simple case of a single 8 byte block that has been definitely lost. The second case mentions another 8 byte block that has been definitely lost; the difference is that a further 80 bytes in other blocks are indirectly lost because of this lost block. The loss records are not presented in any notable order, so the loss record numbers aren't particularly meaningful. The loss record numbers can be used in the Valgrind gdbserver to list the addresses of the leaked blocks and/or give more details about how a block is still reachable.

The option `--show-leak-kinds=<set>` controls the set of leak kinds to show when `--leak-check=full` is specified.

The `<set>` of leak kinds is specified in one of the following ways:

- a comma separated list of one or more of `definite indirect possible reachable`.
- `all` to specify the complete set (all leak kinds).
- `none` for the empty set.

The default value for the leak kinds to show is `--show-leak-kinds=definite,possible`.

To also show the reachable and indirectly lost blocks in addition to the definitely and possibly lost blocks, you can use `--show-leak-kinds=all`. To only show the reachable and indirectly lost blocks, use `--show-leak-kinds=indirect,reachable`. The reachable and indirectly lost blocks will then be presented as shown in the following two examples.

```
64 bytes in 4 blocks are still reachable in loss record 2 of 4
   at 0x........: malloc (vg_replace_malloc.c:177)
   by 0x........: mk (leak-cases.c:52)
   by 0x........: main (leak-cases.c:74)

32 bytes in 2 blocks are indirectly lost in loss record 1 of 4
   at 0x........: malloc (vg_replace_malloc.c:177)
   by 0x........: mk (leak-cases.c:52)
   by 0x........: main (leak-cases.c:80)
```

Because there are different kinds of leaks with different severities, an interesting question is: which leaks should be counted as true "errors" and which should not?

The answer to this question affects the numbers printed in the `ERROR SUMMARY` line, and also the effect of the `--error-exitcode` option. First, a leak is only counted as a true "error" if `--leak-check=full` is specified.

Then, the option `--errors-for-leak-kinds=<set>` controls the set of leak kinds to consider as errors. The default value is `--errors-for-leak-kinds=definite,possible`

4.3. Memcheck Command-Line Options

`--leak-check=<no|summary|yes|full> [default: summary]`

When enabled, search for memory leaks when the client program finishes. If set to `summary`, it says how many leaks occurred. If set to `full` or `yes`, each individual leak will be shown in detail and/or counted as an error, as specified by the options `--show-leak-kinds` and `--errors-for-leak-kinds`.

If `--xml=yes` is given, memcheck will automatically use the value `--leak-check=full`. You can use `--show-leak-kinds=none` to reduce the size of the xml output if you are not interested in the leak results.

`--leak-resolution=<low|med|high> [default: high]`

When doing leak checking, determines how willing Memcheck is to consider different backtraces to be the same for the purposes of merging multiple leaks into a single leak report. When set to `low`, only the first two entries need match. When `med`, four entries have to match. When `high`, all entries need to match.

For hardcore leak debugging, you probably want to use `--leak-resolution=high` together with `--num-callers=40` or some such large number.

Note that the `--leak-resolution` setting does not affect Memcheck's ability to find leaks. It only changes how the results are presented.

`--show-leak-kinds=<set> [default: definite,possible]`

Specifies the leak kinds to show in a `full` leak search, in one of the following ways:

- a comma separated list of one or more of `definite indirect possible reachable`.
- `all` to specify the complete set (all leak kinds). It is equivalent to `--show-leak-kinds=definite,indirect,possible,reachable`.
- `none` for the empty set.

`--errors-for-leak-kinds=<set> [default: definite,possible]`

Specifies the leak kinds to count as errors in a `full` leak search. The `<set>` is specified similarly to `--show-leak-kinds`

`--leak-check-heuristics=<set> [default: all]`

Specifies the set of leak check heuristics to be used during leak searches. The heuristics control which interior pointers to a block cause it to be considered as reachable. The heuristic set is specified in one of the following ways:

- a comma separated list of one or more of `stdstring length64 newarray multipleinheritance`.
- `all` to activate the complete set of heuristics. It is equivalent to `--leak-check-heuristics=stdstring,length64,newarray,multipleinheritance`.
- `none` for the empty set.

Note that these heuristics are dependent on the layout of the objects produced by the C++ compiler. They have been tested with some gcc versions (e.g. 4.4 and 4.7). They might not work properly with other C++ compilers.

`--show-reachable=<yes|no> , --show-possibly-lost=<yes|no>`

These options provide an alternative way to specify the leak kinds to show:

- `--show-reachable=no --show-possibly-lost=yes` is equivalent to `--show-leak-kinds=definite,possible`.
- `--show-reachable=no --show-possibly-lost=no` is equivalent to `--show-leak-kinds=definite`.
- `--show-reachable=yes` is equivalent to `--show-leak-kinds=all`.

Note that `--show-possibly-lost=no` has no effect if `--show-reachable=yes` is specified.

`--xtree-leak=<no|yes> [no]`

If set to yes, the results for the leak search done at exit will be output in a 'Callgrind Format' execution tree file. Note that this automatically sets the options `--leak-check=full` and `--show-leak-kinds=all`, to allow xtree visualisation tools such as kcachegrind to select what kind to leak to visualize. The produced file will contain the following events:

- `RB` : Reachable Bytes
- `PB` : Possibly lost Bytes
- `IB` : Indirectly lost Bytes
- `DB` : Definitely lost Bytes (direct plus indirect)
- `DIB` : Definitely Indirectly lost Bytes (subset of DB)
- `RBk` : reachable Blocks
- `PBk` : Possibly lost Blocks
- `IBk` : Indirectly lost Blocks
- `DBk` : Definitely lost Blocks

The increase or decrease for all events above will also be output in the file to provide the delta (increase or decrease) between 2 successive leak searches. For example, `iRB` is the increase of the `RB` event, `dPBk` is the decrease of `PBk` event. The values for the increase and decrease events will be zero for the first leak search done.

See Execution Trees for a detailed explanation about execution trees.

`--xtree-leak-file=<filename> [default: xtleak.kcg.%p]`

Specifies that Valgrind should produce the xtree leak report in the specified file. Any `%p`, `%q` or `%n` sequences appearing in the filename are expanded in exactly the same way as they are for `--log-file`. See the description of --log-file for details.

See Execution Trees for a detailed explanation about execution trees formats.

`--undef-value-errors=<yes|no> [default: yes]`

Controls whether Memcheck reports uses of undefined value errors. Set this to `no` if you don't want to see undefined value errors. It also has the side effect of speeding up Memcheck somewhat. AddrCheck (removed in Valgrind 3.1.0) functioned like Memcheck with `--undef-value-errors=no`.

`--track-origins=<yes|no> [default: no]`

Controls whether Memcheck tracks the origin of uninitialised values. By default, it does not, which means that although it can tell you that an uninitialised value is being used in a dangerous way, it cannot tell you where the uninitialised value came from. This often makes it difficult to track down the root problem.

When set to `yes`, Memcheck keeps track of the origins of all uninitialised values. Then, when an uninitialised value error is reported, Memcheck will try to show the origin of the value. An origin can be one of the following four places: a heap block, a stack allocation, a client request, or miscellaneous other sources (eg, a call to `brk`).

For uninitialised values originating from a heap block, Memcheck shows where the block was allocated. For uninitialised values originating from a stack allocation, Memcheck can tell you which function allocated the value, but no more than that -- typically it shows you the source location of the opening brace of the function. So you should carefully check that all of the function's local variables are initialised properly.

Performance overhead: origin tracking is expensive. It halves Memcheck's speed and increases memory use by a minimum of 100MB, and possibly more. Nevertheless it can drastically reduce the effort required to identify the root cause of uninitialised value errors, and so is often a programmer productivity win, despite running more slowly.

Accuracy: Memcheck tracks origins quite accurately. To avoid very large space and time overheads, some approximations are made. It is possible, although unlikely, that Memcheck will report an incorrect origin, or not be able to identify any origin.

Note that the combination `--track-origins=yes` and `--undef-value-errors=no` is nonsensical. Memcheck checks for and rejects this combination at startup.

`--partial-loads-ok=<yes|no> [default: yes]`

Controls how Memcheck handles 32-, 64-, 128- and 256-bit naturally aligned loads from addresses for which some bytes are addressable and others are not. When `yes`, such loads do not produce an address error. Instead, loaded bytes originating from illegal addresses are marked as uninitialised, and those corresponding to legal addresses are handled in the normal way.

When `no`, loads from partially invalid addresses are treated the same as loads from completely invalid addresses: an illegal-address error is issued, and the resulting bytes are marked as initialised.

Note that code that behaves in this way is in violation of the ISO C/C++ standards, and should be considered broken. If at all possible, such code should be fixed.

`--expensive-definedness-checks=<no|auto|yes> [default: auto]`

Controls whether Memcheck should employ more precise but also more expensive (time consuming) instrumentation when checking the definedness of certain values. In particular, this affects the instrumentation of integer adds, subtracts and equality comparisons.

Selecting `--expensive-definedness-checks=yes` causes Memcheck to use the most accurate analysis possible. This minimises false error rates but can cause up to 30% performance degradation.

Selecting `--expensive-definedness-checks=no` causes Memcheck to use the cheapest instrumentation possible. This maximises performance but will normally give an unusably high false error rate.

The default setting, `--expensive-definedness-checks=auto`, is strongly recommended. This causes Memcheck to use the minimum of expensive instrumentation needed to achieve the same false error rate as `--expensive-definedness-checks=yes`. It also enables an instrumentation-time analysis pass which aims to further reduce the costs of accurate instrumentation. Overall, the performance loss is generally around 5% relative to `--expensive-definedness-checks=no`, although this is strongly workload dependent. Note that the exact instrumentation settings in this mode are architecture dependent.

`--keep-stacktraces=alloc|free|alloc-and-free|alloc-then-free|none [default: alloc-and-free]`

Controls which stack trace(s) to keep for malloc'd and/or free'd blocks.

With `alloc-then-free`, a stack trace is recorded at allocation time, and is associated with the block. When the block is freed, a second stack trace is recorded, and this replaces the allocation stack trace. As a result, any "use after free" errors relating to this block can only show a stack trace for where the block was freed.

With `alloc-and-free`, both allocation and the deallocation stack traces for the block are stored. Hence a "use after free" error will show both, which may make the error easier to diagnose. Compared to `alloc-then-free`, this setting slightly increases Valgrind's memory use as the block contains two references instead of one.

With `alloc`, only the allocation stack trace is recorded (and reported). With `free`, only the deallocation stack trace is recorded (and reported). These values somewhat decrease Valgrind's memory and cpu usage. They can be useful depending on the error types you are searching for and the level of detail you need to analyse them. For example, if you are only interested in memory leak errors, it is sufficient to record the allocation stack traces.

With `none`, no stack traces are recorded for malloc and free operations. If your program allocates a lot of blocks and/or allocates/frees from many different stack traces, this can significantly decrease cpu and/or memory required. Of course, few details will be reported for errors related to heap blocks.

Note that once a stack trace is recorded, Valgrind keeps the stack trace in memory even if it is not referenced by any block. Some programs (for example, recursive algorithms) can generate a huge number of stack traces. If Valgrind uses too much memory in such circumstances, you can reduce the memory required with the options `--keep-stacktraces` and/or by using a smaller value for the option `--num-callers`.

If you want to use `--xtree-memory=full` memory profiling (see Execution Trees), then you cannot specify `--keep-stacktraces=free` or `--keep-stacktraces=none`.

`--freelist-vol=<number> [default: 20000000]`

When the client program releases memory using `free` (in `C`) or `delete` (`C++`), that memory is not immediately made available for re-allocation. Instead, it is marked inaccessible and placed in a queue of freed blocks. The purpose is to defer as long as possible the point at which freed-up memory comes back into circulation. This increases the chance that Memcheck will be able to detect invalid accesses to blocks for some significant period of time after they have been freed.

This option specifies the maximum total size, in bytes, of the blocks in the queue. The default value is twenty million bytes. Increasing this increases the total amount of memory used by Memcheck but may detect invalid uses of freed blocks which would otherwise go undetected.

`--freelist-big-blocks=<number> [default: 1000000]`

When making blocks from the queue of freed blocks available for re-allocation, Memcheck will in priority re-circulate the blocks with a size greater or equal to `--freelist-big-blocks`. This ensures that freeing big blocks (in particular freeing blocks bigger than `--freelist-vol`) does not immediately lead to a re-circulation of all (or a lot of) the small blocks in the free list. In other words, this option increases the likelihood to discover dangling pointers for the "small" blocks, even when big blocks are freed.

Setting a value of 0 means that all the blocks are re-circulated in a FIFO order.

`--workaround-gcc296-bugs=<yes|no> [default: no]`

When enabled, assume that reads and writes some small distance below the stack pointer are due to bugs in GCC 2.96, and does not report them. The "small distance" is 256 bytes by default. Note that GCC 2.96 is the default compiler on some ancient Linux distributions (RedHat 7.X) and so you may need to use this option. Do not use it if you do not have to, as it can cause real errors to be overlooked. A better alternative is to use a more recent GCC in which this bug is fixed.

You may also need to use this option when working with GCC 3.X or 4.X on 32-bit PowerPC Linux. This is because GCC generates code which occasionally accesses below the stack pointer, particularly for floating-point to/from integer conversions. This is in violation of the 32-bit PowerPC ELF specification, which makes no provision for locations below the stack pointer to be accessible.

This option is deprecated as of version 3.12 and may be removed from future versions. You should instead use `--ignore-range-below-sp` to specify the exact range of offsets below the stack pointer that should be ignored. A suitable equivalent is `--ignore-range-below-sp=1024-1`.

`--ignore-range-below-sp=<number>-<number>`

This is a more general replacement for the deprecated `--workaround-gcc296-bugs` option. When specified, it causes Memcheck not to report errors for accesses at the specified offsets below the stack pointer. The two offsets must be positive decimal numbers and -- somewhat counterintuitively -- the first one must be larger, in order to imply a non-wraparound address range to ignore. For example, to ignore 4 byte accesses at 8192 bytes below the stack pointer, use `--ignore-range-below-sp=8192-8189`. Only one range may be specified.

`--show-mismatched-frees=<yes|no> [default: yes]`

When enabled, Memcheck checks that heap blocks are deallocated using a function that matches the allocating function. That is, it expects `free` to be used to deallocate blocks allocated by `malloc`, `delete` for blocks allocated by `new`, and `delete[]` for blocks allocated by `new[]`. If a mismatch is detected, an error is reported. This is in general important because in some environments, freeing with a non-matching function can cause crashes.

There is however a scenario where such mismatches cannot be avoided. That is when the user provides implementations of `new`/`new[]` that call `malloc` and of `delete`/`delete[]` that call `free`, and these functions are asymmetrically inlined. For example, imagine that `delete[]` is inlined but `new[]` is not. The result is that Memcheck "sees" all `delete[]` calls as direct calls to `free`, even when the program source contains no mismatched calls.

This causes a lot of confusing and irrelevant error reports. `--show-mismatched-frees=no` disables these checks. It is not generally advisable to disable them, though, because you may miss real errors as a result.

`--show-realloc-size-zero=<yes|no> [default: yes]`

When enabled, Memcheck checks for uses of `realloc` with a size of zero. This usage of `realloc` is unsafe since it is not portable. On some systems it will behave like `free`. On other systems it will either do nothing or else behave like a call to `free` followed by a call to `malloc` with a size of zero.

`--ignore-ranges=0xPP-0xQQ[,0xRR-0xSS]`

Any ranges listed in this option (and multiple ranges can be specified, separated by commas) will be ignored by Memcheck's addressability checking.

`--malloc-fill=<hexnumber>`

Fills blocks allocated by `malloc`, `new`, etc, but not by `calloc`, with the specified byte. This can be useful when trying to shake out obscure memory corruption problems. The allocated area is still regarded by Memcheck as undefined -- this option only affects its contents. Note that `--malloc-fill` does not affect a block of memory when it is used as argument to client requests VALGRIND_MEMPOOL_ALLOC or VALGRIND_MALLOCLIKE_BLOCK.

`--free-fill=<hexnumber>`

Fills blocks freed by `free`, `delete`, etc, with the specified byte value. This can be useful when trying to shake out obscure memory corruption problems. The freed area is still regarded by Memcheck as not valid for access -- this option only affects its contents. Note that `--free-fill` does not affect a block of memory when it is used as argument to client requests VALGRIND_MEMPOOL_FREE or VALGRIND_FREELIKE_BLOCK.

4.4. Writing suppression files

The basic suppression format is described in Suppressing errors.

The suppression-type (second) line should have the form:

```
Memcheck:suppression_type
```

The Memcheck suppression types are as follows:

- `Value1`, `Value2`, `Value4`, `Value8`, `Value16`, meaning an uninitialised-value error when using a value of 1, 2, 4, 8 or 16 bytes.
- `Cond` (or its old name, `Value0`), meaning use of an uninitialised CPU condition code.
- `Addr1`, `Addr2`, `Addr4`, `Addr8`, `Addr16`, meaning an invalid address during a memory access of 1, 2, 4, 8 or 16 bytes respectively.
- `Jump`, meaning an jump to an unaddressable location error.
- `Param`, meaning an invalid system call parameter error.
- `Free`, meaning an invalid or mismatching free.
- `Overlap`, meaning a `src` / `dst` overlap in `memcpy` or a similar function.
- `Leak`, meaning a memory leak.

`Param` errors have a mandatory extra information line at this point, which is the name of the offending system call parameter.

`Leak` errors have an optional extra information line, with the following format:

```
match-leak-kinds:<set>
```

where `<set>` specifies which leak kinds are matched by this suppression entry. `<set>` is specified in the same way as with the option `--show-leak-kinds`, that is, one of the following:

- a comma separated list of one or more of `definite indirect possible reachable`.
- `all` to specify the complete set (all leak kinds).
- `none` for the empty set.

If this optional extra line is not present, the suppression entry will match all leak kinds.

Be aware that leak suppressions that are created using `--gen-suppressions` will contain this optional extra line, and therefore may match fewer leaks than you expect. You may want to remove the line before using the generated suppressions.

The other Memcheck error kinds do not have extra lines.

If you give the `-v` option, Valgrind will print the list of used suppressions at the end of execution. For a leak suppression, this output gives the number of different loss records that match the suppression, and the number of bytes and blocks suppressed by the suppression. If the run contains multiple leak checks, the number of bytes and blocks are reset to zero before each new leak check. Note that the number of different loss records is not reset to zero.

In the example below, in the last leak search, 7 blocks and 96 bytes have been suppressed by a suppression with the name some_leak_suppression:

```
--21041-- used_suppression:     10 some_other_leak_suppression s.supp:14 suppressed: 12,40
--21041-- used_suppression:     39 some_leak_suppression s.supp:2 suppressed: 96 bytes in
```

For ValueN and AddrN errors, the first line of the calling context is either the name of the function in which the error occurred, or, failing that, the full path of the .so file or executable containing the error location. For Free errors, the first line is the name of the function doing the freeing (eg, free, __builtin_vec_delete, etc). For Overlap errors, the first line is the name of the function with the overlapping arguments (eg. memcpy, strcpy, etc).

The last part of any suppression specifies the rest of the calling context that needs to be matched.

4.5. Details of Memcheck's checking machinery

Read this section if you want to know, in detail, exactly what and how Memcheck is checking.

4.5.1. Valid-value (V) bits

It is simplest to think of Memcheck implementing a synthetic CPU which is identical to a real CPU, except for one crucial detail. Every bit (literally) of data processed, stored and handled by the real CPU has, in the synthetic CPU, an associated "valid-value" bit, which says whether or not the accompanying bit has a legitimate value. In the discussions which follow, this bit is referred to as the V (valid-value) bit.

Each byte in the system therefore has a 8 V bits which follow it wherever it goes. For example, when the CPU loads a word-size item (4 bytes) from memory, it also loads the corresponding 32 V bits from a bitmap which stores the V bits for the process' entire address space. If the CPU should later write the whole or some part of that value to memory at a different address, the relevant V bits will be stored back in the V-bit bitmap.

In short, each bit in the system has (conceptually) an associated V bit, which follows it around everywhere, even inside the CPU. Yes, all the CPU's registers (integer, floating point, vector and condition registers) have their own V bit vectors. For this to work, Memcheck uses a great deal of compression to represent the V bits compactly.

Copying values around does not cause Memcheck to check for, or report on, errors. However, when a value is used in a way which might conceivably affect your program's externally-visible behaviour, the associated V bits are immediately checked. If any of these indicate that the value is undefined (even partially), an error is reported.

Here's an (admittedly nonsensical) example:

```
int i, j;
int a[10], b[10];
for ( i = 0; i < 10; i++ ) {
  j = a[i];
  b[i] = j;
}
```

Memcheck emits no complaints about this, since it merely copies uninitialised values from a[] into b[], and doesn't use them in a way which could affect the behaviour of the program. However, if the loop is changed to:

```
for ( i = 0; i < 10; i++ ) {
  j += a[i];
}
```

```
if ( j == 77 )
  printf("hello there\n");
```

then Memcheck will complain, at the `if`, that the condition depends on uninitialised values. Note that it **doesn't** complain at the `j += a[i];`, since at that point the undefinedness is not "observable". It's only when a decision has to be made as to whether or not to do the `printf` -- an observable action of your program -- that Memcheck complains.

Most low level operations, such as adds, cause Memcheck to use the V bits for the operands to calculate the V bits for the result. Even if the result is partially or wholly undefined, it does not complain.

Checks on definedness only occur in three places: when a value is used to generate a memory address, when control flow decision needs to be made, and when a system call is detected, Memcheck checks definedness of parameters as required.

If a check should detect undefinedness, an error message is issued. The resulting value is subsequently regarded as well-defined. To do otherwise would give long chains of error messages. In other words, once Memcheck reports an undefined value error, it tries to avoid reporting further errors derived from that same undefined value.

This sounds overcomplicated. Why not just check all reads from memory, and complain if an undefined value is loaded into a CPU register? Well, that doesn't work well, because perfectly legitimate C programs routinely copy uninitialised values around in memory, and we don't want endless complaints about that. Here's the canonical example. Consider a struct like this:

```

struct S { int x; char c; };
struct S s1, s2;
s1.x = 42;
s1.c = 'z';
s2 = s1;
```

The question to ask is: how large is `struct S`, in bytes? An `int` is 4 bytes and a `char` one byte, so perhaps a `struct S` occupies 5 bytes? Wrong. All non-toy compilers we know of will round the size of `struct S` up to a whole number of words, in this case 8 bytes. Not doing this forces compilers to generate truly appalling code for accessing arrays of `struct S`'s on some architectures.

So `s1` occupies 8 bytes, yet only 5 of them will be initialised. For the assignment `s2 = s1`, GCC generates code to copy all 8 bytes wholesale into `s2` without regard for their meaning. If Memcheck simply checked values as they came out of memory, it would yelp every time a structure assignment like this happened. So the more complicated behaviour described above is necessary. This allows GCC to copy `s1` into `s2` any way it likes, and a warning will only be emitted if the uninitialised values are later used.

As explained above, Memcheck maintains 8 V bits for each byte in your process, including for bytes that are in shared memory. However, the same piece of shared memory can be mapped multiple times, by several processes or even by the same process (for example, if the process wants a read-only and a read-write mapping of the same page). For such multiple mappings, Memcheck tracks the V bits for each mapping independently. This can lead to false positive errors, as the shared memory can be initialised via a first mapping, and accessed via another mapping. The access via this other mapping will have its own V bits, which have not been changed when the memory was initialised via the first mapping. The bypass for these false positives is to use Memcheck's client requests `VALGRIND_MAKE_MEM_DEFINED` and `VALGRIND_MAKE_MEM_UNDEFINED` to inform Memcheck about what your program does (or what another process does) to these shared memory mappings.

4.5.2. Valid-address (A) bits

Notice that the previous subsection describes how the validity of values is established and maintained without having to say whether the program does or does not have the right to access any particular memory location. We now consider the latter question.

As described above, every bit in memory or in the CPU has an associated valid-value (V) bit. In addition, all bytes in memory, but not in the CPU, have an associated valid-address (A) bit. This indicates whether or not the program can legitimately read or write that location. It does not give any indication of the validity of the data at that location -- that's the job of the V bits -- only whether or not the location may be accessed.

Every time your program reads or writes memory, Memcheck checks the A bits associated with the address. If any of them indicate an invalid address, an error is emitted. Note that the reads and writes themselves do not change the A bits, only consult them.

So how do the A bits get set/cleared? Like this:

- When the program starts, all the global data areas are marked as accessible.
- When the program does `malloc`/`new`, the A bits for exactly the area allocated, and not a byte more, are marked as accessible. Upon freeing the area the A bits are changed to indicate inaccessibility.
- When the stack pointer register (`SP`) moves up or down, A bits are set. The rule is that the area from `SP` up to the base of the stack is marked as accessible, and below `SP` is inaccessible. (If that sounds illogical, bear in mind that the stack grows down, not up, on almost all Unix systems, including GNU/Linux.) Tracking `SP` like this has the useful side-effect that the section of stack used by a function for local variables etc is automatically marked accessible on function entry and inaccessible on exit.
- When doing system calls, A bits are changed appropriately. For example, `mmap` magically makes files appear in the process' address space, so the A bits must be updated if `mmap` succeeds.
- Optionally, your program can tell Memcheck about such changes explicitly, using the client request mechanism described above.

4.5.3. Putting it all together

Memcheck's checking machinery can be summarised as follows:

- Each byte in memory has 8 associated V (valid-value) bits, saying whether or not the byte has a defined value, and a single A (valid-address) bit, saying whether or not the program currently has the right to read/write that address. As mentioned above, heavy use of compression means the overhead is typically around 25%.
- When memory is read or written, the relevant A bits are consulted. If they indicate an invalid address, Memcheck emits an Invalid read or Invalid write error.
- When memory is read into the CPU's registers, the relevant V bits are fetched from memory and stored in the simulated CPU. They are not consulted.
- When a register is written out to memory, the V bits for that register are written back to memory too.
- When values in CPU registers are used to generate a memory address, or to determine the outcome of a conditional branch, the V bits for those values are checked, and an error emitted if any of them are undefined.
- When values in CPU registers are used for any other purpose, Memcheck computes the V bits for the result, but does not check them.
- Once the V bits for a value in the CPU have been checked, they are then set to indicate validity. This avoids long chains of errors.
- When values are loaded from memory, Memcheck checks the A bits for that location and issues an illegal-address warning if needed. In that case, the V bits loaded are forced to indicate Valid, despite the location being invalid.

  This apparently strange choice reduces the amount of confusing information presented to the user. It avoids the unpleasant phenomenon in which memory is read from a place which is both unaddressable and contains invalid values, and, as a result, you get not only an invalid-address (read/write) error, but also a potentially large set of uninitialised-value errors, one for every time the value is used.

There is a hazy boundary case to do with multi-byte loads from addresses which are partially valid and partially invalid. See details of the option `--partial-loads-ok` for details.

Memcheck intercepts calls to `malloc`, `calloc`, `realloc`, `valloc`, `memalign`, `free`, `new`, `new[]`, `delete` and `delete[]`. The behaviour you get is:

- `malloc`/`new`/`new[]`: the returned memory is marked as addressable but not having valid values. This means you have to write to it before you can read it.
- `calloc`: returned memory is marked both addressable and valid, since `calloc` clears the area to zero.
- `realloc`: if the new size is larger than the old, the new section is addressable but invalid, as with `malloc`. If the new size is smaller, the dropped-off section is marked as unaddressable. You may only pass to `realloc` a pointer previously issued to you by `malloc`/`calloc`/`realloc`.
- `free`/`delete`/`delete[]`: you may only pass to these functions a pointer previously issued to you by the corresponding allocation function. Otherwise, Memcheck complains. If the pointer is indeed valid, Memcheck marks the entire area it points at as unaddressable, and places the block in the freed-blocks-queue. The aim is to defer as long as possible reallocation of this block. Until that happens, all attempts to access it will elicit an invalid-address error, as you would hope.

4.6. Memcheck Monitor Commands

The Memcheck tool provides monitor commands handled by Valgrind's built-in gdbserver (see Monitor command handling by the Valgrind gdbserver). Valgrind python code provides GDB front end commands giving an easier usage of the memcheck monitor commands (see GDB front end commands for Valgrind gdbserver monitor commands). To launch a memcheck monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `memcheck` command (or the shorter aliases `mc`). Using the memcheck GDB front end command provide a more flexible usage, such as evaluation of address and length arguments by GDB. In GDB, you can use `help memcheck` to get help about the memcheck front end monitor commands and you can use `apropos memcheck` to get all the commands mentionning the word "memcheck" in their name or on-line help.

- `xb <addr> [<len>]` shows the definedness (V) bits and values for <len> (default 1) bytes starting at <addr>. For each 8 bytes, two lines are output.

  The first line shows the validity bits for 8 bytes. The definedness of each byte in the range is given using two hexadecimal digits. These hexadecimal digits encode the validity of each bit of the corresponding byte, using 0 if the bit is defined and 1 if the bit is undefined. If a byte is not addressable, its validity bits are replaced by __ (a double underscore).

  The second line shows the values of the bytes below the corresponding validity bits. The format used to show the bytes data is similar to the GDB command 'x /<len>xb <addr>'. The value for a non addressable bytes is shown as ?? (two question marks).

  In the following example, `string10` is an array of 10 characters, in which the even numbered bytes are undefined. In the below example, the byte corresponding to `string10[5]` is not addressable.

```
(gdb) p &string10
$4 = (char (*)[10]) 0x804a2f0
(gdb) mo xb 0x804a2f0 10
                  ff       00       ff       00       ff       __       ff       00
0x804A2F0:      0x3f     0x6e     0x3f     0x65     0x3f     0x??     0x3f     0x65
```

```
                  ff      00
0x804A2F8:      0x3f    0x00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
(gdb)
```

The GDB memcheck front end command `memcheck xb ADDR [LEN]` accepts any address expression for its first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space. The following example shows how to get the definedness of `string10` using the memcheck xb front end command.

```
(gdb) mc xb &string10 sizeof(string10)
                  ff      00      ff      00      ff      __      ff      00
0x804A2F0:      0x3f    0x6e    0x3f    0x65    0x3f    0x??    0x3f    0x65
                  ff      00
0x804A2F8:      0x3f    0x00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
(gdb)
```

The command xb cannot be used with registers. To get the validity bits of a register, you must start Valgrind with the option `--vgdb-shadow-registers=yes`. The validity bits of a register can then be obtained by printing the 'shadow 1' corresponding register. In the below x86 example, the register eax has all its bits undefined, while the register ebx is fully defined.

```
(gdb) p /x $eaxs1
$9 = 0xffffffff
(gdb) p /x $ebxs1
$10 = 0x0
(gdb)
```

- `get_vbits <addr> [<len>]` shows the definedness (V) bits for <len> (default 1) bytes starting at <addr> using the same convention as the `xb` command. `get_vbits` only shows the V bits (grouped by 4 bytes). It does not show the values. If you want to associate V bits with the corresponding byte values, the `xb` command will be easier to use, in particular on little endian computers when associating undefined parts of an integer with their V bits values.

  The following example shows the result of `get_vbits` on the `string10` used in the `xb` command explanation. The GDB memcheck equivalent front end command `memcheck get_vbits ADDR [LEN]` accepts any ADDR expression and any LEN expression (separated by a space).

```
(gdb) monitor get_vbits 0x804a2f0 10
ff00ff00 ff__ff00 ff00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
(gdb) memcheck get_vbits &string10 sizeof(string10)
ff00ff00 ff__ff00 ff00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
```

- `make_memory [noaccess|undefined|defined|Definedifaddressable] <addr> [<len>]` marks the range of <len> (default 1) bytes at <addr> as having the given status. Parameter `noaccess` marks the range as non-accessible, so Memcheck will report an error on any access to it. `undefined` or `defined` mark the area as accessible, but Memcheck regards the bytes in it respectively as having undefined or defined values. `Definedifaddressable` marks as defined, bytes in the range which are already addressible, but makes no change to the status of bytes in the range which are not addressible. Note that the first letter of `Definedifaddressable` is an uppercase D to avoid confusion with `defined`.

The GDB equivalent memcheck front end commands `memcheck make_memory [noaccess|undefined|defined|Definedifaddressable] ADDR [LEN]` accept any address expression for their first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

In the following example, the first byte of the `string10` is marked as defined and then is marked noaccess:

```
(gdb) monitor make_memory defined 0x8049e28  1
(gdb) monitor get_vbits 0x8049e28 10
0000ff00 ff00ff00 ff00
(gdb) memcheck make_memory noaccess &string10[0]
(gdb) memcheck get_vbits &string10 sizeof(string10)
__00ff00 ff00ff00 ff00
Address 0x8049E28 len 10 has 1 bytes unaddressable
(gdb)
```

- `check_memory [addressable|defined] <addr> [<len>]` checks that the range of <len> (default 1) bytes at <addr> has the specified accessibility. It then outputs a description of <addr>. In the following example, a detailed description is available because the option `--read-var-info=yes` was given at Valgrind startup:

```
(gdb) monitor check_memory defined 0x8049e28  1
Address 0x8049E28 len 1 defined
==14698==  Location 0x8049e28 is 0 bytes inside string10[0],
==14698==  declared at prog.c:10, in frame #0 of thread 1
(gdb)
```

  The GDB equivalent memcheck front end commands `memcheck check_memory [addressable|defined] ADDR [LEN]` accept any address expression for their first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

- `leak_check [full*|summary|xtleak] [kinds <set>|reachable|possibleleak*|definiteleak] [heuristics heur1,heur2,...] [new|increased*|changed|any] [unlimited*|limited <max_loss_records_output>]` performs a leak check. The `*` in the arguments indicates the default values.

  If the `[full*|summary|xtleak]` argument is `summary`, only a summary of the leak search is given; otherwise a full leak report is produced. A full leak report gives detailed information for each leak: the stack trace where the leaked blocks were allocated, the number of blocks leaked and their total size. When a full report is requested, the next two arguments further specify what kind of leaks to report. A leak's details are shown if they match both the second and third argument. A full leak report might output detailed information for many leaks. The nr of leaks for which information is output can be controlled using the `limited` argument followed by the maximum nr of leak records to output. If this maximum is reached, the leak search outputs the records with the biggest number of bytes.

  The value `xtleak` also produces a full leak report, but output it as an xtree in a file xtleak.kcg.%p.%n (see --log-file). See Execution Trees for a detailed explanation about execution trees formats. See --xtree-leak for the description of the events in a xtree leak file.

  The `kinds` argument controls what kind of blocks are shown for a `full` leak search. The set of leak kinds to show can be specified using a `<set>` similarly to the command line option `--show-leak-kinds`. Alternatively, the value `definiteleak` is equivalent to `kinds definite`, the value `possibleleak` is equivalent to `kinds definite,possible` : it will also show possibly leaked blocks, .i.e those for which only an interior pointer was found. The value `reachable` will show all block categories (i.e. is equivalent to `kinds all`).

The `heuristics` argument controls the heuristics used during the leak search. The set of heuristics to use can be specified using a `<set>` similarly to the command line option `--leak-check-heuristics`. The default value for the `heuristics` argument is `heuristics none`.

The `[new|increased*|changed|any]` argument controls what kinds of changes are shown for a `full` leak search. The value `increased` specifies that only block allocation stacks with an increased number of leaked bytes or blocks since the previous leak check should be shown. The value `changed` specifies that allocation stacks with any change since the previous leak check should be shown. The value `new` specifies to show only the block allocation stacks that are new since the previous leak search. The value `any` specifies that all leak entries should be shown, regardless of any increase or decrease. If `new` or `increased` or `changed` are specified, the leak report entries will show the delta relative to the previous leak report and the new loss records will have a "new" marker (even when `increased` or `changed` were specified).

The following example shows usage of the `leak_check` monitor command on the `memcheck/tests/leak-cases.c` regression test. The first command outputs one entry having an increase in the leaked bytes. The second command is the same as the first command, but uses the abbreviated forms accepted by GDB and the Valgrind gdbserver. It only outputs the summary information, as there was no increase since the previous leak search.

```
(gdb) monitor leak_check full possibleleak increased
==19520== 16 (+16) bytes in 1 (+1) blocks are possibly lost in new loss record 9 of 12
==19520==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19520==    by 0x80484D5: mk (leak-cases.c:52)
==19520==    by 0x804855F: f (leak-cases.c:81)
==19520==    by 0x80488E0: main (leak-cases.c:107)
==19520==
==19520== LEAK SUMMARY:
==19520==    definitely lost: 32 (+0) bytes in 2 (+0) blocks
==19520==    indirectly lost: 16 (+0) bytes in 1 (+0) blocks
==19520==      possibly lost: 32 (+16) bytes in 2 (+1) blocks
==19520==    still reachable: 96 (+16) bytes in 6 (+1) blocks
==19520==         suppressed: 0 (+0) bytes in 0 (+0) blocks
==19520== Reachable blocks (those to which a pointer was found) are not shown.
==19520== To see them, add 'reachable any' args to leak_check
==19520==
(gdb) mo l
==19520== LEAK SUMMARY:
==19520==    definitely lost: 32 (+0) bytes in 2 (+0) blocks
==19520==    indirectly lost: 16 (+0) bytes in 1 (+0) blocks
==19520==      possibly lost: 32 (+0) bytes in 2 (+0) blocks
==19520==    still reachable: 96 (+0) bytes in 6 (+0) blocks
==19520==         suppressed: 0 (+0) bytes in 0 (+0) blocks
==19520== Reachable blocks (those to which a pointer was found) are not shown.
==19520== To see them, add 'reachable any' args to leak_check
==19520==
(gdb)
```

Note that when using Valgrind's gdbserver, it is not necessary to rerun with `--leak-check=full --show-reachable=yes` to see the reachable blocks. You can obtain the same information without rerunning by using the GDB command `monitor leak_check full reachable any` (or, using abbreviation: `mo l f r a`).

The GDB equivalent memcheck front end command `memcheck leak_check` auto-completes the user input by providing the full list of keywords still relevant according to what is already typed. For example, if the "summary" keyword has been provided, the following TABs to auto-complete other items will not propose anymore "full" and "xtleak". Note that KIND and HEUR values are not part of auto-completed elements.

- `block_list <loss_record_nr>|<loss_record_nr_from>..<loss_record_nr_to> [unlimited*|limited <max_blocks>] [heuristics heur1,heur2,...]` shows the list of blocks belonging to `<loss_record_nr>` (or to the loss records range `<loss_record_nr_from>..<loss_record_nr_to>`). The nr of blocks to print can be controlled using the `limited` argument followed by the maximum nr of blocks to output. If one or more heuristics are given, only prints the loss records and blocks found via one of the given `heur1,heur2,...` heuristics.

  A leak search merges the allocated blocks in loss records : a loss record re-groups all blocks having the same state (for example, Definitely Lost) and the same allocation backtrace. Each loss record is identified in the leak search result by a loss record number. The `block_list` command shows the loss record information followed by the addresses and sizes of the blocks which have been merged in the loss record. If a block was found using an heuristic, the block size is followed by the heuristic.

  If a directly lost block causes some other blocks to be indirectly lost, the block_list command will also show these indirectly lost blocks. The indirectly lost blocks will be indented according to the level of indirection between the directly lost block and the indirectly lost block(s). Each indirectly lost block is followed by the reference of its loss record.

  The block_list command can be used on the results of a leak search as long as no block has been freed after this leak search: as soon as the program frees a block, a new leak search is needed before block_list can be used again.

  In the below example, the program leaks a tree structure by losing the pointer to the block A (top of the tree). So, the block A is directly lost, causing an indirect loss of blocks B to G. The first block_list command shows the loss record of A (a definitely lost block with address 0x4028028, size 16). The addresses and sizes of the indirectly lost blocks due to block A are shown below the block A. The second command shows the details of one of the indirect loss records output by the first command.

```
           A
         /   \
        B     C
       / \   / \
      D   E F   G
```

```
(gdb) bt
#0  main () at leak-tree.c:69
(gdb) monitor leak_check full any
==19552== 112 (16 direct, 96 indirect) bytes in 1 blocks are definitely lost in loss reco
==19552==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19552==    by 0x80484D5: mk (leak-tree.c:28)
==19552==    by 0x80484FC: f (leak-tree.c:41)
==19552==    by 0x8048856: main (leak-tree.c:63)
==19552==
==19552== LEAK SUMMARY:
==19552==    definitely lost: 16 bytes in 1 blocks
==19552==    indirectly lost: 96 bytes in 6 blocks
==19552==      possibly lost: 0 bytes in 0 blocks
==19552==    still reachable: 0 bytes in 0 blocks
==19552==         suppressed: 0 bytes in 0 blocks
==19552==
(gdb) monitor block_list 7
```

```
==19552== 112 (16 direct, 96 indirect) bytes in 1 blocks are definitely lost in loss reco
==19552==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19552==    by 0x80484D5: mk (leak-tree.c:28)
==19552==    by 0x80484FC: f (leak-tree.c:41)
==19552==    by 0x8048856: main (leak-tree.c:63)
==19552== 0x4028028[16]
==19552==   0x4028068[16] indirect loss record 1
==19552==      0x40280E8[16] indirect loss record 3
==19552==      0x4028128[16] indirect loss record 4
==19552==   0x40280A8[16] indirect loss record 2
==19552==      0x4028168[16] indirect loss record 5
==19552==      0x40281A8[16] indirect loss record 6
(gdb) mo b 2
==19552== 16 bytes in 1 blocks are indirectly lost in loss record 2 of 7
==19552==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19552==    by 0x80484D5: mk (leak-tree.c:28)
==19552==    by 0x8048519: f (leak-tree.c:43)
==19552==    by 0x8048856: main (leak-tree.c:63)
==19552== 0x40280A8[16]
==19552==   0x4028168[16] indirect loss record 5
==19552==   0x40281A8[16] indirect loss record 6
(gdb)
```

- `who_points_at <addr> [<len>]` shows all the locations where a pointer to addr is found. If len is equal to 1, the command only shows the locations pointing exactly at addr (i.e. the "start pointers" to addr). If len is > 1, "interior pointers" pointing at the len first bytes will also be shown.

  The locations searched for are the same as the locations used in the leak search. So, `who_points_at` can a.o. be used to show why the leak search still can reach a block, or can search for dangling pointers to a freed block. Each location pointing at addr (or pointing inside addr if interior pointers are being searched for) will be described.

  The GDB equivalent memcheck front end command `memcheck who_points_at ADDR [LEN]` accept any address expression for its first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

  In the below example, the pointers to the 'tree block A' (see example in command `block_list`) is shown before the tree was leaked. The descriptions are detailed as the option `--read-var-info=yes` was given at Valgrind startup. The second call shows the pointers (start and interior pointers) to block G. The block G (0x40281A8) is reachable via block C (0x40280a8) and register ECX of tid 1 (tid is the Valgrind thread id). It is "interior reachable" via the register EBX.

```
(gdb) monitor who_points_at 0x4028028
==20852== Searching for pointers to 0x4028028
==20852== *0x8049e20 points at 0x4028028
==20852==  Location 0x8049e20 is 0 bytes inside global var "t"
==20852==  declared at leak-tree.c:35
(gdb) monitor who_points_at 0x40281A8 16
==20852== Searching for pointers pointing in 16 bytes from 0x40281a8
==20852== *0x40280ac points at 0x40281a8
==20852==  Address 0x40280ac is 4 bytes inside a block of size 16 alloc'd
==20852==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==20852==    by 0x80484D5: mk (leak-tree.c:28)
==20852==    by 0x8048519: f (leak-tree.c:43)
```

```
==20852==    by 0x8048856: main (leak-tree.c:63)
==20852== tid 1 register ECX points at 0x40281a8
==20852== tid 1 register EBX interior points at 2 bytes inside 0x40281a8
(gdb)
```

When `who_points_at` finds an interior pointer, it will report the heuristic(s) with which this interior pointer will be considered as reachable. Note that this is done independently of the value of the option `--leak-check-heuristics`. In the below example, the loss record 6 indicates a possibly lost block. `who_points_at` reports that there is an interior pointer pointing in this block, and that the block can be considered reachable using the heuristic `multipleinheritance`.

```
(gdb) monitor block_list 6
==3748== 8 bytes in 1 blocks are possibly lost in loss record 6 of 7
==3748==    at 0x4007D77: operator new(unsigned int) (vg_replace_malloc.c:313)
==3748==    by 0x8048954: main (leak_cpp_interior.cpp:43)
==3748== 0x402A0E0[8]
(gdb) monitor who_points_at 0x402A0E0 8
==3748== Searching for pointers pointing in 8 bytes from 0x402a0e0
==3748== *0xbe8ee078 interior points at 4 bytes inside 0x402a0e0
==3748==  Address 0xbe8ee078 is on thread 1's stack
==3748== block at 0x402a0e0 considered reachable by ptr 0x402a0e4 using multipleinherita
(gdb)
```

- `xtmemory [<filename> default xtmemory.kcg.%p.%n]` requests Memcheck tool to produce an xtree heap memory report. See Execution Trees for a detailed explanation about execution trees.

4.7. Client Requests

The following client requests are defined in `memcheck.h`. See `memcheck.h` for exact details of their arguments.

- `VALGRIND_MAKE_MEM_NOACCESS`, `VALGRIND_MAKE_MEM_UNDEFINED` and `VALGRIND_MAKE_MEM_DEFINED`. These mark address ranges as completely inaccessible, accessible but containing undefined data, and accessible and containing defined data, respectively. They return -1, when run on Valgrind and 0 otherwise.

- `VALGRIND_MAKE_MEM_DEFINED_IF_ADDRESSABLE`. This is just like `VALGRIND_MAKE_MEM_DEFINED` but only affects those bytes that are already addressable.

- `VALGRIND_CHECK_MEM_IS_ADDRESSABLE` and `VALGRIND_CHECK_MEM_IS_DEFINED`: check immediately whether or not the given address range has the relevant property, and if not, print an error message. Also, for the convenience of the client, returns zero if the relevant property holds; otherwise, the returned value is the address of the first byte for which the property is not true. Always returns 0 when not run on Valgrind.

- `VALGRIND_CHECK_VALUE_IS_DEFINED`: a quick and easy way to find out whether Valgrind thinks a particular value (lvalue, to be precise) is addressable and defined. Prints an error message if not. It has no return value.

- `VALGRIND_DO_LEAK_CHECK`: does a full memory leak check (like `--leak-check=full`) right now. This is useful for incrementally checking for leaks between arbitrary places in the program's execution. It has no return value.

- `VALGRIND_DO_ADDED_LEAK_CHECK`: same as `VALGRIND_DO_LEAK_CHECK` but only shows the entries for which there was an increase in leaked bytes or leaked number of blocks since the previous leak search. It has no return value.

- `VALGRIND_DO_CHANGED_LEAK_CHECK`: same as `VALGRIND_DO_LEAK_CHECK` but only shows the entries for which there was an increase or decrease in leaked bytes or leaked number of blocks since the previous leak search. It has no return value.

- `VALGRIND_DO_NEW_LEAK_CHECK`: same as `VALGRIND_DO_LEAK_CHECK` but only shows the new entries since the previous leak search. It has no return value.

- `VALGRIND_DO_QUICK_LEAK_CHECK`: like `VALGRIND_DO_LEAK_CHECK`, except it produces only a leak summary (like `--leak-check=summary`). It has no return value.

- `VALGRIND_COUNT_LEAKS`: fills in the four arguments with the number of bytes of memory found by the previous leak check to be leaked (i.e. the sum of direct leaks and indirect leaks), dubious, reachable and suppressed. This is useful in test harness code, after calling `VALGRIND_DO_LEAK_CHECK` or `VALGRIND_DO_QUICK_LEAK_CHECK`.

- `VALGRIND_COUNT_LEAK_BLOCKS`: identical to `VALGRIND_COUNT_LEAKS` except that it returns the number of blocks rather than the number of bytes in each category.

- `VALGRIND_GET_VBITS` and `VALGRIND_SET_VBITS`: allow you to get and set the V (validity) bits for an address range. You should probably only set V bits that you have got with `VALGRIND_GET_VBITS`. Only for those who really know what they are doing.

- `VALGRIND_CREATE_BLOCK` and `VALGRIND_DISCARD`. `VALGRIND_CREATE_BLOCK` takes an address, a number of bytes and a character string. The specified address range is then associated with that string. When Memcheck reports an invalid access to an address in the range, it will describe it in terms of this block rather than in terms of any other block it knows about. Note that the use of this macro does not actually change the state of memory in any way -- it merely gives a name for the range.

  At some point you may want Memcheck to stop reporting errors in terms of the block named by `VALGRIND_CREATE_BLOCK`. To make this possible, `VALGRIND_CREATE_BLOCK` returns a "block handle", which is a C `int` value. You can pass this block handle to `VALGRIND_DISCARD`. After doing so, Valgrind will no longer relate addressing errors in the specified range to the block. Passing invalid handles to `VALGRIND_DISCARD` is harmless.

4.8. Memory Pools: describing and working with custom allocators

Some programs use custom memory allocators, often for performance reasons. Left to itself, Memcheck is unable to understand the behaviour of custom allocation schemes as well as it understands the standard allocators, and so may miss errors and leaks in your program. What this section describes is a way to give Memcheck enough of a description of your custom allocator that it can make at least some sense of what is happening.

There are many different sorts of custom allocator, so Memcheck attempts to reason about them using a loose, abstract model. We use the following terminology when describing custom allocation systems:

- Custom allocation involves a set of independent "memory pools".

- Memcheck's notion of a a memory pool consists of a single "anchor address" and a set of non-overlapping "chunks" associated with the anchor address.

- Typically a pool's anchor address is the address of a book-keeping "header" structure.

- Typically the pool's chunks are drawn from a contiguous "superblock" acquired through the system `malloc` or `mmap`.

Keep in mind that the last two points above say "typically": the Valgrind mempool client request API is intentionally vague about the exact structure of a mempool. There is no specific mention made of headers or superblocks. Nevertheless, the following picture may help elucidate the intention of the terms in the API:

```
"pool"
(anchor address)
|
v
+--------+---+
| header | o |
+--------+-|-+
           |
           v                  superblock
           +------+---+--------------+---+------------------+
           |      |rzB|  allocation  |rzB|                  |
           +------+---+--------------+---+------------------+
                      ^              ^
                      |              |
                    "addr"     "addr"+"size"
```

Note that the header and the superblock may be contiguous or discontiguous, and there may be multiple superblocks associated with a single header; such variations are opaque to Memcheck. The API only requires that your allocation scheme can present sensible values of "pool", "addr" and "size".

Typically, before making client requests related to mempools, a client program will have allocated such a header and superblock for their mempool, and marked the superblock NOACCESS using the `VALGRIND_MAKE_MEM_NOACCESS` client request.

When dealing with mempools, the goal is to maintain a particular invariant condition: that Memcheck believes the unallocated portions of the pool's superblock (including redzones) are NOACCESS. To maintain this invariant, the client program must ensure that the superblock starts out in that state; Memcheck cannot make it so, since Memcheck never explicitly learns about the superblock of a pool, only the allocated chunks within the pool.

Once the header and superblock for a pool are established and properly marked, there are a number of client requests programs can use to inform Memcheck about changes to the state of a mempool:

- `VALGRIND_CREATE_MEMPOOL(pool, rzB, is_zeroed)`: This request registers the address `pool` as the anchor address for a memory pool. It also provides a size `rzB`, specifying how large the redzones placed around chunks allocated from the pool should be. Finally, it provides an `is_zeroed` argument that specifies whether the pool's chunks are zeroed (more precisely: defined) when allocated.

  Upon completion of this request, no chunks are associated with the pool. The request simply tells Memcheck that the pool exists, so that subsequent calls can refer to it as a pool.

- `VALGRIND_CREATE_MEMPOOL_EXT(pool, rzB, is_zeroed, flags)`: Create a memory pool with some flags (that can be OR-ed together) specifying extended behaviour. When flags is zero, the behaviour is identical to `VALGRIND_CREATE_MEMPOOL`.

  - The flag `VALGRIND_MEMPOOL_METAPOOL` specifies that the pieces of memory associated with the pool using `VALGRIND_MEMPOOL_ALLOC` will be used by the application as superblocks to dole out MALLOC_LIKE blocks using `VALGRIND_MALLOCLIKE_BLOCK`. In other words, a meta pool is a "2 levels" pool : first level is the blocks described by `VALGRIND_MEMPOOL_ALLOC`. The second level blocks are described using `VALGRIND_MALLOCLIKE_BLOCK`. Note that the association between the pool and the second level blocks is implicit : second level blocks will be located inside first level blocks. It is necessary to use the

VALGRIND_MEMPOOL_METAPOOL flag for such 2 levels pools, as otherwise valgrind will detect overlapping memory blocks, and will abort execution (e.g. during leak search).

- VALGRIND_MEMPOOL_AUTO_FREE. Such a meta pool can also be marked as an 'auto free' pool using the flag VALGRIND_MEMPOOL_AUTO_FREE, which must be OR-ed together with the VALGRIND_MEMPOOL_METAPOOL. For an 'auto free' pool, VALGRIND_MEMPOOL_FREE will automatically free the second level blocks that are contained inside the first level block freed with VALGRIND_MEMPOOL_FREE. In other words, calling VALGRIND_MEMPOOL_FREE will cause implicit calls to VALGRIND_FREELIKE_BLOCK for all the second level blocks included in the first level block. Note: it is an error to use the VALGRIND_MEMPOOL_AUTO_FREE flag without the VALGRIND_MEMPOOL_METAPOOL flag.

- VALGRIND_DESTROY_MEMPOOL(pool): This request tells Memcheck that a pool is being torn down. Memcheck then removes all records of chunks associated with the pool, as well as its record of the pool's existence. While destroying its records of a mempool, Memcheck resets the redzones of any live chunks in the pool to NOACCESS.

- VALGRIND_MEMPOOL_ALLOC(pool, addr, size): This request informs Memcheck that a size-byte chunk has been allocated at addr, and associates the chunk with the specified pool. If the pool was created with nonzero rzB redzones, Memcheck will mark the rzB bytes before and after the chunk as NOACCESS. If the pool was created with the is_zeroed argument set, Memcheck will mark the chunk as DEFINED, otherwise Memcheck will mark the chunk as UNDEFINED.

- VALGRIND_MEMPOOL_FREE(pool, addr): This request informs Memcheck that the chunk at addr should no longer be considered allocated. Memcheck will mark the chunk associated with addr as NOACCESS, and delete its record of the chunk's existence.

- VALGRIND_MEMPOOL_TRIM(pool, addr, size): This request trims the chunks associated with pool. The request only operates on chunks associated with pool. Trimming is formally defined as:

  - All chunks entirely inside the range addr..(addr+size-1) are preserved.

  - All chunks entirely outside the range addr..(addr+size-1) are discarded, as though VALGRIND_MEMPOOL_FREE was called on them.

  - All other chunks must intersect with the range addr..(addr+size-1); areas outside the intersection are marked as NOACCESS, as though they had been independently freed with VALGRIND_MEMPOOL_FREE.

  This is a somewhat rare request, but can be useful in implementing the type of mass-free operations common in custom LIFO allocators.

- VALGRIND_MOVE_MEMPOOL(poolA, poolB): This request informs Memcheck that the pool previously anchored at address poolA has moved to anchor address poolB. This is a rare request, typically only needed if you realloc the header of a mempool.

  No memory-status bits are altered by this request.

- VALGRIND_MEMPOOL_CHANGE(pool, addrA, addrB, size): This request informs Memcheck that the chunk previously allocated at address addrA within pool has been moved and/or resized, and should be changed to cover the region addrB..(addrB+size-1). This is a rare request, typically only needed if you realloc a superblock or wish to extend a chunk without changing its memory-status bits.

  No memory-status bits are altered by this request.

- VALGRIND_MEMPOOL_EXISTS(pool): This request informs the caller whether or not Memcheck is currently tracking a mempool at anchor address pool. It evaluates to 1 when there is a mempool associated with that address, 0 otherwise. This is a rare request, only useful in circumstances when client code might have lost track of the set of active mempools.

4.9. Debugging MPI Parallel Programs with Valgrind

Memcheck supports debugging of distributed-memory applications which use the MPI message passing standard. This support consists of a library of wrapper functions for the `PMPI_*` interface. When incorporated into the application's address space, either by direct linking or by `LD_PRELOAD`, the wrappers intercept calls to `PMPI_Send`, `PMPI_Recv`, etc. They then use client requests to inform Memcheck of memory state changes caused by the function being wrapped. This reduces the number of false positives that Memcheck otherwise typically reports for MPI applications.

The wrappers also take the opportunity to carefully check size and definedness of buffers passed as arguments to MPI functions, hence detecting errors such as passing undefined data to `PMPI_Send`, or receiving data into a buffer which is too small.

Unlike most of the rest of Valgrind, the wrapper library is subject to a BSD-style license, so you can link it into any code base you like. See the top of `mpi/libmpiwrap.c` for license details.

4.9.1. Building and installing the wrappers

The wrapper library will be built automatically if possible. Valgrind's configure script will look for a suitable `mpicc` to build it with. This must be the same `mpicc` you use to build the MPI application you want to debug. By default, Valgrind tries `mpicc`, but you can specify a different one by using the configure-time option `--with-mpicc`. Currently the wrappers are only buildable with `mpicc`s which are based on GNU GCC or Intel's C++ Compiler.

Check that the configure script prints a line like this:

```
checking for usable MPI2-compliant mpicc and mpi.h... yes, mpicc
```

If it says `... no`, your `mpicc` has failed to compile and link a test MPI2 program.

If the configure test succeeds, continue in the usual way with `make` and `make install`. The final install tree should then contain `libmpiwrap-<platform>.so`.

Compile up a test MPI program (eg, MPI hello-world) and try this:

```
LD_PRELOAD=$prefix/lib/valgrind/libmpiwrap-<platform>.so   \
           mpirun [args] $prefix/bin/valgrind ./hello
```

You should see something similar to the following

```
valgrind MPI wrappers 31901: Active for pid 31901
valgrind MPI wrappers 31901: Try MPIWRAP_DEBUG=help for possible options
```

repeated for every process in the group. If you do not see these, there is an build/installation problem of some kind.

The MPI functions to be wrapped are assumed to be in an ELF shared object with soname matching `libmpi.so*`. This is known to be correct at least for Open MPI and Quadrics MPI, and can easily be changed if required.

4.9.2. Getting started

Compile your MPI application as usual, taking care to link it using the same mpicc that your Valgrind build was configured with.

Use the following basic scheme to run your application on Valgrind with the wrappers engaged:

```
MPIWRAP_DEBUG=[wrapper-args]                                  \
   LD_PRELOAD=$prefix/lib/valgrind/libmpiwrap-<platform>.so   \
   mpirun [mpirun-args]                                       \
   $prefix/bin/valgrind [valgrind-args]                       \
   [application] [app-args]
```

As an alternative to LD_PRELOADing libmpiwrap-<platform>.so, you can simply link it to your application if desired. This should not disturb native behaviour of your application in any way.

4.9.3. Controlling the wrapper library

Environment variable MPIWRAP_DEBUG is consulted at startup. The default behaviour is to print a starting banner

```
valgrind MPI wrappers 16386: Active for pid 16386
valgrind MPI wrappers 16386: Try MPIWRAP_DEBUG=help for possible options
```

and then be relatively quiet.

You can give a list of comma-separated options in MPIWRAP_DEBUG. These are

- verbose: show entries/exits of all wrappers. Also show extra debugging info, such as the status of outstanding MPI_Requests resulting from uncompleted MPI_Irecvs.
- quiet: opposite of verbose, only print anything when the wrappers want to report a detected programming error, or in case of catastrophic failure of the wrappers.
- warn: by default, functions which lack proper wrappers are not commented on, just silently ignored. This causes a warning to be printed for each unwrapped function used, up to a maximum of three warnings per function.
- strict: print an error message and abort the program if a function lacking a wrapper is used.

If you want to use Valgrind's XML output facility (--xml=yes), you should pass quiet in MPIWRAP_DEBUG so as to get rid of any extraneous printing from the wrappers.

4.9.4. Functions

All MPI2 functions except MPI_Wtick, MPI_Wtime and MPI_Pcontrol have wrappers. The first two are not wrapped because they return a double, which Valgrind's function-wrap mechanism cannot handle (but it could easily be extended to do so). MPI_Pcontrol cannot be wrapped as it has variable arity: int MPI_Pcontrol(const int level, ...)

Most functions are wrapped with a default wrapper which does nothing except complain or abort if it is called, depending on settings in MPIWRAP_DEBUG listed above. The following functions have "real", do-something-useful wrappers:

```
PMPI_Send PMPI_Bsend PMPI_Ssend PMPI_Rsend

PMPI_Recv PMPI_Get_count

PMPI_Isend PMPI_Ibsend PMPI_Issend PMPI_Irsend

PMPI_Irecv
PMPI_Wait PMPI_Waitall
PMPI_Test PMPI_Testall

PMPI_Iprobe PMPI_Probe

PMPI_Cancel

PMPI_Sendrecv

PMPI_Type_commit PMPI_Type_free

PMPI_Pack PMPI_Unpack

PMPI_Bcast PMPI_Gather PMPI_Scatter PMPI_Alltoall
PMPI_Reduce PMPI_Allreduce PMPI_Op_create

PMPI_Comm_create PMPI_Comm_dup PMPI_Comm_free PMPI_Comm_rank PMPI_Comm_size

PMPI_Error_string
PMPI_Init PMPI_Initialized PMPI_Finalize
```

A few functions such as `PMPI_Address` are listed as `HAS_NO_WRAPPER`. They have no wrapper at all as there is nothing worth checking, and giving a no-op wrapper would reduce performance for no reason.

Note that the wrapper library itself can itself generate large numbers of calls to the MPI implementation, especially when walking complex types. The most common functions called are `PMPI_Extent`, `PMPI_Type_get_envelope`, `PMPI_Type_get_contents`, and `PMPI_Type_free`.

4.9.5. Types

MPI-1.1 structured types are supported, and walked exactly. The currently supported combiners are `MPI_COMBINER_NAMED`, `MPI_COMBINER_CONTIGUOUS`, `MPI_COMBINER_VECTOR`, `MPI_COMBINER_HVECTOR` `MPI_COMBINER_INDEXED`, `MPI_COMBINER_HINDEXED` and `MPI_COMBINER_STRUCT`. This should cover all MPI-1.1 types. The mechanism (function `walk_type`) should extend easily to cover MPI2 combiners.

MPI defines some named structured types (`MPI_FLOAT_INT`, `MPI_DOUBLE_INT`, `MPI_LONG_INT`, `MPI_2INT`, `MPI_SHORT_INT`, `MPI_LONG_DOUBLE_INT`) which are pairs of some basic type and a C `int`. Unfortunately the MPI specification makes it impossible to look inside these types and see where the fields are. Therefore these wrappers assume the types are laid out as `struct { float val; int loc; }` (for `MPI_FLOAT_INT`), etc, and act accordingly. This appears to be correct at least for Open MPI 1.0.2 and for Quadrics MPI.

If `strict` is an option specified in `MPIWRAP_DEBUG`, the application will abort if an unhandled type is encountered. Otherwise, the application will print a warning message and continue.

Some effort is made to mark/check memory ranges corresponding to arrays of values in a single pass. This is important for performance since asking Valgrind to mark/check any range, no matter how small, carries quite a large constant cost. This optimisation is applied to arrays of primitive types (`double`, `float`, `int`, `long`, `long long`, `short`, `char`, and `long double` on platforms where `sizeof(long double) == 8`). For arrays of all other types, the wrappers handle each element individually and so there can be a very large performance cost.

4.9.6. Writing new wrappers

For the most part the wrappers are straightforward. The only significant complexity arises with nonblocking receives.

The issue is that `MPI_Irecv` states the recv buffer and returns immediately, giving a handle (`MPI_Request`) for the transaction. Later the user will have to poll for completion with `MPI_Wait` etc, and when the transaction completes successfully, the wrappers have to paint the recv buffer. But the recv buffer details are not presented to `MPI_Wait` -- only the handle is. The library therefore maintains a shadow table which associates uncompleted `MPI_Requests` with the corresponding buffer address/count/type. When an operation completes, the table is searched for the associated address/count/type info, and memory is marked accordingly.

Access to the table is guarded by a (POSIX pthreads) lock, so as to make the library thread-safe.

The table is allocated with `malloc` and never `freed`, so it will show up in leak checks.

Writing new wrappers should be fairly easy. The source file is `mpi/libmpiwrap.c`. If possible, find an existing wrapper for a function of similar behaviour to the one you want to wrap, and use it as a starting point. The wrappers are organised in sections in the same order as the MPI 1.1 spec, to aid navigation. When adding a wrapper, remember to comment out the definition of the default wrapper in the long list of defaults at the bottom of the file (do not remove it, just comment it out).

4.9.7. What to expect when using the wrappers

The wrappers should reduce Memcheck's false-error rate on MPI applications. Because the wrapping is done at the MPI interface, there will still potentially be a large number of errors reported in the MPI implementation below the interface. The best you can do is try to suppress them.

You may also find that the input-side (buffer length/definedness) checks find errors in your MPI use, for example passing too short a buffer to `MPI_Recv`.

Functions which are not wrapped may increase the false error rate. A possible approach is to run with `MPI_DEBUG` containing `warn`. This will show you functions which lack proper wrappers but which are nevertheless used. You can then write wrappers for them.

A known source of potential false errors are the `PMPI_Reduce` family of functions, when using a custom (user-defined) reduction function. In a reduction operation, each node notionally sends data to a "central point" which uses the specified reduction function to merge the data items into a single item. Hence, in general, data is passed between nodes and fed to the reduction function, but the wrapper library cannot mark the transferred data as initialised before it is handed to the reduction function, because all that happens "inside" the `PMPI_Reduce` call. As a result you may see false positives reported in your reduction function.

5. Cachegrind: a high-precision tracing profiler

To use this tool, specify `--tool=cachegrind` on the Valgrind command line.

5.1. Overview

Cachegrind is a high-precision tracing profiler. It runs slowly, but collects precise and reproducible profiling data. It can merge and diff data from different runs. To expand on these characteristics:

- *Precise.* Cachegrind measures the exact number of instructions executed by your program, not an approximation. Furthermore, it presents the gathered data at the file, function, and line level. This is different to many other profilers that measure approximate execution time, using sampling, and only at the function level.

- *Reproducible.* In general, execution time is a better metric than instruction counts because it's what users perceive. However, execution time often has high variability. When running the exact same program on the exact same input multiple times, execution time might vary by several percent. Furthermore, small changes in a program can change its memory layout and have even larger effects on runtime. In contrast, instruction counts are highly reproducible; for some programs they are perfectly reproducible. This means the effects of small changes in a program can be measured with high precision.

For these reasons, Cachegrind is an excellent complement to time-based profilers.

Cachegrind can annotate programs written in any language, so long as debug info is present to map machine code back to the original source code. Cachegrind has been used successfully on programs written in C, C++, Rust, and assembly.

Cachegrind can also simulate how your program interacts with a machine's cache hierarchy and branch predictor. This simulation was the original motivation for the tool, hence its name. However, the simulations are basic and unlikely to reflect the behaviour of a modern machine. For this reason they are off by default. If you really want cache and branch information, a profiler like `perf` that accesses hardware counters is a better choice.

5.2. Using Cachegrind and cg_annotate

First, as for normal Valgrind use, you should compile with debugging info (the `-g` option in most compilers). But by contrast with normal Valgrind use, you probably do want to turn optimisation on, since you should profile your program as it will be normally run.

Second, run Cachegrind itself to gather the profiling data.

Third, run cg_annotate to get a detailed presentation of that data. cg_annotate can combine the results of multiple Cachegrind output files. It can also perform a diff between two Cachegrind output files.

5.2.1. Running Cachegrind

To run Cachegrind on a program `prog`, run:

```
valgrind --tool=cachegrind prog
```

The program will execute (slowly). Upon completion, summary statistics that look like this will be printed:

```
==17942== I refs:          8,195,070
```

The `I refs` number is short for "Instruction cache references", which is equivalent to "instructions executed". If you enable the cache and/or branch simulation, additional counts will be shown.

5.2.2. Output File

Cachegrind also writes more detailed profiling data to a file. By default this Cachegrind output file is named `cachegrind.out.<pid>` (where <pid> is the program's process ID), but its name can be changed with the `--cachegrind-out-file` option. This file is human-readable, but is intended to be interpreted by the accompanying program cg_annotate, described in the next section.

The default `.<pid>` suffix on the output file name serves two purposes. First, it means existing Cachegrind output files aren't immediately overwritten. Second, and more importantly, it allows correct profiling with the `--trace-children=yes` option of programs that spawn child processes.

5.2.3. Running cg_annotate

Before using cg_annotate, it is worth widening your window to be at least 120 characters wide if possible, because the output lines can be quite long.

Then run:

```
cg_annotate <filename>
```

on a Cachegrind output file.

5.2.4. The Metadata Section

The first part of the output looks like this:

```
--------------------------------------------------------------------------------
-- Metadata
--------------------------------------------------------------------------------
Invocation:       ../cg_annotate concord.cgout
Command:          ./concord ../cg_main.c
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Threshold:        0.1%
Annotation:       on
```

It summarizes how Cachegrind and the profiled program were run.

- Invocation: the command line used to produce this output.
- Command: the command line used to run the profiled program.
- Events recorded: which events were recorded. By default, this is `Ir`. More events will be recorded if cache and/or branch simulation is enabled.
- Events shown: the events shown, which is a subset of the events gathered. This can be adjusted with the `--show` option.

- Event sort order: the sort order used for the subsequent sections. For example, in this case those sections are sorted from highest `Ir` counts to lowest. If there are multiple events, one will be the primary sort event, and then there can be a secondary sort event, tertiary sort event, etc., though more than one is rarely needed. This order can be adjusted with the `--sort` option. Note that this does *not* specify the order in which the columns appear. That is specified by the "events shown" line (and can be changed with the `--show` option).
- Threshold: cg_annotate by default omits files and functions with very low counts to keep the output size reasonable. By default cg_annotate only shows files and functions that account for at least 0.1% of the primary sort event. The threshold can be adjusted with the `--threshold` option.
- Annotation: whether source file annotation is enabled. Controlled with the `--annotate` option.

If cache simulation is enabled, details of the cache parameters will be shown above the "Invocation" line.

5.2.5. Global, File, and Function-level Counts

Next comes the summary for the whole program:

```
--------------------------------------------------------------------------------
-- Summary
--------------------------------------------------------------------------------
Ir_______________

8,195,070 (100.0%)  PROGRAM TOTALS
```

The `Ir` column label is suffixed with underscores to show the bounds of the columns underneath.

Then comes file:function counts. Here is the first part of that section:

```
--------------------------------------------------------------------------------
-- File:function summary
--------------------------------------------------------------------------------
  Ir______________________  file:function

< 3,078,746 (37.6%, 37.6%)  /home/njn/grind/ws1/cachegrind/concord.c:
  1,630,232 (19.9%)           get_word
    630,918  (7.7%)           hash
    461,095  (5.6%)           insert
    130,560  (1.6%)           add_existing
     91,014  (1.1%)           init_hash_table
     88,056  (1.1%)           create
     46,676  (0.6%)           new_word_node

< 1,746,038 (21.3%, 58.9%)  ./malloc/./malloc/malloc.c:
  1,285,938 (15.7%)           _int_malloc
    458,225  (5.6%)           malloc

< 1,107,550 (13.5%, 72.4%)  ./libio/./libio/getc.c:getc

<   551,071  (6.7%, 79.1%)  ./string/../sysdeps/x86_64/multiarch/strcmp-avx2.S:__strcmp_av

<   521,228  (6.4%, 85.5%)  ./ctype/../include/ctype.h:
```

```
    260,616  (3.2%)           __ctype_tolower_loc
    260,612  (3.2%)           __ctype_b_loc

<   468,163  (5.7%, 91.2%)  ???:
    468,151  (5.7%)           ???

<   456,071  (5.6%, 96.8%)  /usr/include/ctype.h:get_word
```

Each entry covers one file, and one or more functions within that file. If there is only one significant function within a file, as in the first entry, the file and function are shown on the same line separate by a colon. If there are multiple significant functions within a file, as in the third entry, each function gets its own line.

This example involves a small C program, and shows a combination of code from the program itself (including functions like `get_word` and `hash` in the file `concord.c`) as well as code from system libraries, such as functions like `malloc` and `getc`.

Each entry is preceded with a <, which can be useful when navigating through the output in an editor, or grepping through results.

The first percentage in each column indicates the proportion of the total event count is covered by this line. The second percentage, which only shows on the first line of each entry, shows the cumulative percentage of all the entries up to and including this one. The entries shown here account for 96.8% of the instructions executed by the program.

The name `???` is used if the file name and/or function name could not be determined from debugging information. If `???` filenames dominate, the program probably wasn't compiled with `-g`. If `???` function names dominate, the program may have had symbols stripped.

After that comes function:file counts. Here is the first part of that section:

```
--------------------------------------------------------------------------------
-- Function:file summary
--------------------------------------------------------------------------------
  Ir______________________  function:file

> 2,086,303 (25.5%, 25.5%)  get_word:
  1,630,232 (19.9%)           /home/njn/grind/ws1/cachegrind/concord.c
    456,071  (5.6%)           /usr/include/ctype.h

> 1,285,938 (15.7%, 41.1%)  _int_malloc:./malloc/./malloc/malloc.c

> 1,107,550 (13.5%, 54.7%)  getc:./libio/./libio/getc.c

>   630,918  (7.7%, 62.4%)  hash:/home/njn/grind/ws1/cachegrind/concord.c

>   551,071  (6.7%, 69.1%)  __strcmp_avx2:./string/../sysdeps/x86_64/multiarch/strcmp-avx2

>   480,248  (5.9%, 74.9%)  malloc:
    458,225  (5.6%)           ./malloc/./malloc/malloc.c
     22,023  (0.3%)           ./malloc/./malloc/arena.c

>   468,151  (5.7%, 80.7%)  ???:???

>   461,095  (5.6%, 86.3%)  insert:/home/njn/grind/ws1/cachegrind/concord.c
```

This is similar to the previous section, but is grouped by functions first and files second. Also, the entry markers are > instead of <.

You might wonder why this section is needed, and how it differs from the previous section. The answer is inlining. In this example there are two entries demonstrating a function whose code is effectively spread across more than one file: `get_word` and `malloc`. Here is an example from profiling the Rust compiler, a much larger program that uses inlining more:

```
>  30,469,230 (1.3%, 11.1%)  <rustc_middle::ty::context::CtxtInterners>::intern_ty:
   10,269,220 (0.5%)           /home/njn/.cargo/registry/src/github.com-1ecc6299db9ec823/h
    7,696,827 (0.3%)           /home/njn/dev/rust0/compiler/rustc_middle/src/ty/context.rs
    3,858,099 (0.2%)           /home/njn/dev/rust0/library/core/src/cell.rs
```

In this case the compiled function `intern_ty` includes code from three different source files, due to inlining. These should be examined together. Older versions of cg_annotate presented this entry as three separate file:function entries, which would typically be intermixed with all the other entries, making it hard to see that they are all really part of the same function.

5.2.6. Per-line Counts

By default, a source file is annotated if it contains at least one function that meets the significance threshold. This can be disabled with the `--annotate` option.

To continue the previous example, here is part of the annotation of the file `concord.c`:

```
--------------------------------------------------------------------------------
-- Annotated source file: /home/njn/grind/ws1/cachegrind/docs/concord.c
--------------------------------------------------------------------------------
Ir_____________

      .         /* Function builds the hash table from the given file. */
      .         void init_hash_table(char *file_name, Word_Node *table[])
      8 (0.0%)  {
      .             FILE *file_ptr;
      .             Word_Info *data;
      2 (0.0%)      int line = 1, i;
      .
      .             /* Structure used when reading in words and line numbers. */
      3 (0.0%)      data = (Word_Info *) create(sizeof(Word_Info));
      .
      .             /* Initialise entire table to NULL. */
  2,993 (0.0%)      for (i = 0; i < TABLE_SIZE; i++)
    997 (0.0%)          table[i] = NULL;
      .
      .             /* Open file, check it. */
      4 (0.0%)      file_ptr = fopen(file_name, "r");
      2 (0.0%)      if (!(file_ptr)) {
      .                 fprintf(stderr, "Couldn't open '%s'.\n", file_name);
      .                 exit(EXIT_FAILURE);
      .             }
      .
      .             /*  'Get' the words and lines one at a time from the file, and insert
      .             ** into the table one at a time. */
```

```
 55,363 (0.7%)       while ((line = get_word(data, line, file_ptr)) != EOF)
 31,632 (0.4%)           insert(data->word, data->line, table);
      .
      2 (0.0%)       free(data);
      2 (0.0%)       fclose(file_ptr);
      6 (0.0%)   }
```

Each executed line is annotated with its event counts. Other lines are annotated with a dot. This may be because they contain no executable code, or they contain executable code but were never executed.

You can easily tell if a function is inlined from this output. If it is not inlined, it will have event counts on the lines containing the opening and closing braces. If it is inlined, it will not have event counts on those lines. In the example above, `init_hash_table` does have counts, so you can tell it is not inlined.

Note again that inlining can lead to surprising results. If a function `f` is always inlined, in the file:function and function:file sections counts will be attributed to the functions it is inlined into, rather than itself. However, if you look at the line-by-line annotations for `f` you'll see the counts that belong to `f`. So it's worth looking for large counts/percentages in the line-by-line annotations.

Sometimes only a small section of a source file is executed. To minimise uninteresting output, Cachegrind only shows annotated lines and lines within a small distance of annotated lines. Gaps are marked with line numbers, for example:

```

(counts and code for line 704)
-- line 375 ----------------------------------------
-- line 514 ----------------------------------------
(counts and code for line 878)
```

The number of lines of context shown around annotated lines is controlled by the `--context` option.

Any significant source files that could not be found are shown like this:

```

--------------------------------------------------------------------------------
-- Annotated source file: ./malloc/./malloc/malloc.c
--------------------------------------------------------------------------------
Unannotated because one or more of these original files are unreadable:
- ./malloc/./malloc/malloc.c
```

This is common for library files, because libraries are usually compiled with debugging information but the source files are rarely present on a system.

Cachegrind relies heavily on accurate debug info. Sometimes compilers do not map a particular compiled instruction to line number 0, where the 0 represents "unknown" or "none". This is annoying but does happen in practice. cg_annotate prints these in the following way:

```

--------------------------------------------------------------------------------
-- Annotated source file: /home/njn/dev/rust0/compiler/rustc_borrowck/src/lib.rs
--------------------------------------------------------------------------------
Ir________________

1,046,746 (0.0%)  <unknown (line 0)>
```

Finally, when annotation is performed, the output ends with a summary of how many counts were annotated and unannotated, and why. For example:

```
--------------------------------------------------------------------------------
-- Annotation summary
--------------------------------------------------------------------------------
Ir_______________

3,534,817 (43.1%)    annotated: files known & above threshold & readable, line numbers kno
        0            annotated: files known & above threshold & readable, line numbers unk
        0          unannotated: files known & above threshold & two or more non-identical
4,132,126 (50.4%)  unannotated: files known & above threshold & unreadable
   59,950  (0.7%)  unannotated: files known & below threshold
  468,163  (5.7%)  unannotated: files unknown
```

5.2.7. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent.

If the output file name (controlled by `--cachegrind-out-file`) does not contain `%p`, then the outputs from the parent and child will be intermingled in a single output file, which will almost certainly make it unreadable by cg_annotate.

5.2.8. cg_annotate Warnings

There are two situations in which cg_annotate prints warnings.

- If a source file is more recent than the Cachegrind output file. This is because the information in the Cachegrind output file is only recorded with line numbers, so if the line numbers change at all in the source (e.g. lines added, deleted, swapped), any annotations will be incorrect.
- If information is recorded about line numbers past the end of a file. This can be caused by the above problem, e.g. shortening the source file while using an old Cachegrind output file. If this happens, the figures for the bogus lines are printed anyway (and clearly marked as bogus) in case they are important.

5.2.9. Merging Cachegrind Output Files

cg_annotate can merge data from multiple Cachegrind output files in a single run. (There is also a program called cg_merge that can merge multiple Cachegrind output files into a single Cachegrind output file, but it is now deprecated because cg_annotate's merging does a better job.)

Use it as follows:

```
cg_annotate file1 file2 file3 ...
```

cg_annotate computes the sum of these files (effectively `file1` + `file2` + `file3`), and then produces output as usual that shows the summed counts.

The most common merging scenario is if you want to aggregate costs over multiple runs of the same program, possibly on different inputs.

5.2.10. Differencing Cachegrind output files

cg_annotate can diff data from two Cachegrind output files in a single run. (There is also a program called cg_diff that can diff two Cachegrind output files into a single Cachegrind output file, but it is now deprecated because cg_annotate's differencing does a better job.)

Use it as follows:

```
cg_annotate --diff file1 file2
```

cg_annotate computes the difference between these two files (effectively `file2` - `file1`), and then produces output as usual that shows the count differences. Note that many of the counts may be negative; this indicates that the counts for the relevant file/function/line are smaller in the second version than those in the first version.

The simplest common scenario is comparing two Cachegrind output files that came from the same program, but on different inputs. cg_annotate will do a good job on this without assistance.

A more complex scenario is if you want to compare Cachegrind output files from two slightly different versions of a program that you have sitting side-by-side, running on the same input. For example, you might have `version1/prog.c` and `version2/prog.c`. A straight comparison of the two would not be useful. Because functions are always paired with filenames, a function `f` would be listed as `version1/prog.c:f` for the first version but `version2/prog.c:f` for the second version.

In this case, use the `--mod-filename` option. Its argument is a search-and-replace expression that will be applied to all the filenames in both Cachegrind output files. It can be used to remove minor differences in filenames. For example, the option `--mod-filename='s/version[0-9]/versionN/'` will suffice for the above example.

Similarly, sometimes compilers auto-generate certain functions and give them randomized names like `T.1234` where the suffixes vary from build to build. You can use the `--mod-funcname` option to remove small differences like these; it works in the same way as `--mod-filename`.

When `--mod-filename` is used to compare two different versions of the same program, cg_annotate will not annotate any file that is different between the two versions, because the per-line counts are not reliable in such a case. For example, imagine if `version2/prog.c` is the same as `version1/prog.c` except with an extra blank line at the top of the file. Every single per-line count will have changed. In comparison, the per-file and per-function counts have not changed, and are still very useful for determining differences between programs. You might think that this means every interesting file will be left unannotated, but again inlining means that files that are identical in the two versions can have different counts on many lines.

5.2.11. Cache and Branch Simulation

Cachegrind can simulate how your program interacts with a machine's cache hierarchy and/or branch predictor. The cache simulation models a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). For these machines (in the cases where Cachegrind can auto-detect the cache configuration) Cachegrind simulates the first-level and last-level caches. Therefore, Cachegrind always refers to the I1, D1 and LL (last-level) caches.

When simulating the cache, with `--cache-sim=yes`, Cachegrind gathers the following statistics:

- I cache reads (`Ir`, which equals the number of instructions executed), I1 cache read misses (`I1mr`) and LL cache instruction read misses (`ILmr`).
- D cache reads (`Dr`, which equals the number of memory reads), D1 cache read misses (`D1mr`), and LL cache data read misses (`DLmr`).
- D cache writes (`Dw`, which equals the number of memory writes), D1 cache write misses (`D1mw`), and LL cache data write misses (`DLmw`).

Note that D1 total accesses is given by `D1mr` + `D1mw`, and that LL total accesses is given by `ILmr` + `DLmr` + `DLmw`.

When simulating the branch predictor, with `--branch-sim=yes`, Cachegrind gathers the following statistics:

- Conditional branches executed (`Bc`) and conditional branches mispredicted (`Bcm`).
- Indirect branches executed (`Bi`) and indirect branches mispredicted (`Bim`).

When cache and/or branch simulation is enabled, cg_annotate will print multiple counts per line of output. For example:

```
  Ir________________________ Bc_______________________ Bcm_____________________ Bi___________________

>      8,547   (0.1%, 99.4%)      936   (0.1%, 99.1%)     177   (0.3%, 96.7%)        59   (0.0%, 99
       8,503   (0.1%)             928   (0.1%)            175   (0.3%)               59   (0.0%)
```

5.3. Cachegrind Command-line Options

Cachegrind-specific options are:

`--cachegrind-out-file=<file>`

Write the Cachegrind output file to `file` rather than to the default output file, `cachegrind.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

`--cache-sim=no|yes [no]`

Enables or disables collection of cache access and miss counts.

`--branch-sim=no|yes [no]`

Enables or disables collection of branch instruction and misprediction counts.

`--instr-at-start=no|yes [yes]`

Enables or disables instrumentation at the start of execution. Use this in combination with `CACHEGRIND_START_INSTRUMENTATION` and `CACHEGRIND_STOP_INSTRUMENTATION` to measure only part of a client program's execution.

`--I1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 instruction cache. Only useful with `--cache-sim=yes`.

`--D1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 data cache. Only useful with `--cache-sim=yes`.

`--LL=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the last-level cache. Only useful with `--cache-sim=yes`.

5.4. cg_annotate Command-line Options

`-h --help`

Show the help message.

`--version`

Show the version number.

`--diff`

Diff two Cachegrind output files.

`--mod-filename <regex> [default: none]`

Specifies an `s/old/new/` search-and-replace expression that is applied to all filenames. Useful when differencing, for removing minor differences in paths between two different versions of a program that are sitting in different directories. An `i` suffix makes the regex case-insensitive, and a `g` suffix makes it match multiple times.

`--mod-funcname <regex> [default: none]`

Like `--mod-filename`, but for filenames. Useful for removing minor differences in randomized names of auto-generated functions generated by some compilers.

`--show=A,B,C [default: all, using order in the Cachegrind output file]`

Specifies which events to show (and the column order). Default is to use all present in the Cachegrind output file (and use the order in the file). Best used in conjunction with `--sort`.

`--sort=A,B,C [default: order in the Cachegrind output file]`

Specifies the events upon which the sorting of the file:function and function:file entries will be based.

`--threshold=X [default: 0.1%]`

Sets the significance threshold for the file:function and function:files sections. A file or function is shown if it accounts for more than X% of the counts for the primary sort event. If annotating source files, this also affects which files are annotated.

`--show-percs, --no-show-percs, --show-percs=<no|yes> [default: yes]`

When enabled, a percentage is printed next to all event counts. This helps gauge the relative importance of each function and line.

`--annotate, --no-annotate, --auto=<no|yes> [default: yes]`

Enables or disables source file annotation.

`--context=N [default: 8]`

The number of lines of context to show before and after each annotated line. Use a large number (e.g. 100000) to show all source lines.

5.5. cg_merge Command-line Options

`-o outfile`

Write the output to to `outfile` instead of standard output.

5.6. cg_diff Command-line Options

`-h --help`

Show the help message.

`--version`

Show the version number.

`--mod-filename=<expr> [default: none]`

Specifies an `s/old/new/` search-and-replace expression that is applied to all filenames.

`--mod-funcname=<expr> [default: none]`

Like `--mod-filename`, but for filenames.

5.7. Cachegrind Client Requests

Cachegrind provides the following client requests in `cachegrind.h`.

`CACHEGRIND_START_INSTRUMENTATION`

Start Cachegrind instrumentation if not already enabled. Use this in combination with `CACHEGRIND_STOP_INSTRUMENTATION` and `--instr-at-start` to measure only part of a client program's execution.

`CACHEGRIND_STOP_INSTRUMENTATION`

Stop Cachegrind instrumentation if not already disabled. Use this in combination with `CACHEGRIND_START_INSTRUMENTATION` and `--instr-at-start` to measure only part of a client program's execution.

5.8. Simulation Details

This section talks about details you don't need to know about in order to use Cachegrind, but may be of interest to some people.

5.8.1. Cache Simulation Specifics

The cache simulation approximates the hardware of an AMD Athlon CPU circa 2002. Its specific characteristics are as follows:

- Write-allocate: when a write miss occurs, the block written to is brought into the D1 cache. Most modern caches have this property.
- Bit-selection hash function: the set of line(s) in the cache to which a memory block maps is chosen by the middle bits M--(M+N-1) of the byte address, where:
  - line size = 2^M bytes
  - (cache size / line size / associativity) = 2^N bytes
- Inclusive LL cache: the LL cache typically replicates all the entries of the L1 caches, because fetching into L1 involves fetching into LL first (this does not guarantee strict inclusiveness, as lines evicted from LL still could reside in L1). This is standard on Pentium chips, but AMD Opterons, Athlons and Durons use an exclusive LL cache that only holds blocks evicted from L1. Ditto most modern VIA CPUs.

The cache configuration simulated (cache size, associativity and line size) is determined automatically using the x86 CPUID instruction. If you have a machine that (a) doesn't support the CPUID instruction, or (b) supports it in an early incarnation that doesn't give any cache information, then Cachegrind will fall back to using a default configuration (that of a model 3/4 Athlon). Cachegrind will tell you if this happens. You can manually specify one, two or all three levels (I1/D1/LL) of the cache from the command line using the `--I1`, `--D1` and `--LL` options. For cache parameters to be valid for simulation, the number of sets (with associativity being the number of cache lines in each set) has to be a power of two.

On PowerPC platforms Cachegrind cannot automatically determine the cache configuration, so you will need to specify it with the `--I1`, `--D1` and `--LL` options.

Other noteworthy behaviour:

- References that straddle two cache lines are treated as follows:
  - If both blocks hit --> counted as one hit
  - If one block hits, the other misses --> counted as one miss.
  - If both blocks miss --> counted as one miss (not two)
- Instructions that modify a memory location (e.g. `inc` and `dec`) are counted as doing just a read, i.e. a single data reference. This may seem strange, but since the write can never cause a miss (the read guarantees the block is in the cache) it's not very interesting.

  Thus it measures not the number of times the data cache is accessed, but the number of times a data cache miss could occur.

If you are interested in simulating a cache with different properties, it is not particularly hard to write your own cache simulator, or to modify the existing ones in `cg_sim.c`.

5.8.2. Branch Simulation Specifics

Cachegrind simulates branch predictors intended to be typical of mainstream desktop/server processors of around 2004.

Conditional branches are predicted using an array of 16384 2-bit saturating counters. The array index used for a branch instruction is computed partly from the low-order bits of the branch instruction's address and partly using the taken/not-taken behaviour of the last few conditional branches. As a result the predictions for any specific branch depend both on its own history and the behaviour of previous branches. This is a standard technique for improving prediction accuracy.

For indirect branches (that is, jumps to unknown destinations) Cachegrind uses a simple branch target address predictor. Targets are predicted using an array of 512 entries indexed by the low order 9 bits of the branch instruction's address. Each branch is predicted to jump to the same address it did last time. Any other behaviour causes a mispredict.

More recent processors have better branch predictors, in particular better indirect branch predictors. Cachegrind's predictor design is deliberately conservative so as to be representative of the large installed base of processors which pre-date widespread deployment of more sophisticated indirect branch predictors. In particular, late model Pentium 4s (Prescott), Pentium M, Core and Core 2 have more sophisticated indirect branch predictors than modelled by Cachegrind.

Cachegrind does not simulate a return stack predictor. It assumes that processors perfectly predict function return addresses, an assumption which is probably close to being true.

See Hennessy and Patterson's classic text "Computer Architecture: A Quantitative Approach", 4th edition (2007), Section 2.3 (pages 80-89) for background on modern branch predictors.

5.8.3. Accuracy

Cachegrind's instruction counting has one shortcoming on x86/amd64:

- When a `REP`-prefixed instruction executes each iteration is counted separately. In contrast, hardware counters count each such instruction just once, no matter how many times it iterates. It is arguable that Cachegrind's behaviour is more useful.

Cachegrind's cache profiling has a number of shortcomings:

- It doesn't account for kernel activity. The effect of system calls on the cache and branch predictor contents is ignored.
- It doesn't account for other process activity. This is arguably desirable when considering a single program.
- It doesn't account for virtual-to-physical address mappings. Hence the simulation is not a true representation of what's happening in the cache. Most caches and branch predictors are physically indexed, but Cachegrind simulates caches using virtual addresses.
- It doesn't account for cache misses not visible at the instruction level, e.g. those arising from TLB misses, or speculative execution.
- Valgrind will schedule threads differently from how they would be when running natively. This could warp the results for threaded programs.
- The x86/amd64 instructions `bts`, `btr` and `btc` will incorrectly be counted as doing a data read if both the arguments are registers, e.g.:

  ```
  btsl %eax, %edx
  ```

  This should only happen rarely.
- x86/amd64 FPU instructions with data sizes of 28 and 108 bytes (e.g. `fsave`) are treated as though they only access 16 bytes. These instructions seem to be rare so hopefully this won't affect accuracy much.

Another thing worth noting is that results are very sensitive. Changing the size of the executable being profiled, or the sizes of any of the shared libraries it uses, or even the length of their file names, can perturb the results. Variations will be small, but don't expect perfectly repeatable results if your program changes at all.

Many Linux distributions perform address space layout randomisation (ASLR), in which identical runs of the same program have their shared libraries loaded at different locations, as a security measure. This also perturbs the results.

5.9. Implementation Details

This section talks about details you don't need to know about in order to use Cachegrind, but may be of interest to some people.

5.9.1. How Cachegrind Works

The best reference for understanding how Cachegrind works is chapter 3 of "Dynamic Binary Analysis and Instrumentation", by Nicholas Nethercote. It is available on the Valgrind publications page.

5.9.2. Cachegrind Output File Format

The file format is fairly straightforward, basically giving the cost centre for every line, grouped by files and functions. It's also totally generic and self-describing, in the sense that it can be used for any events that can be counted on a line-by-line basis, not just cache and branch predictor events. For example, earlier versions of Cachegrind didn't have a branch predictor simulation. When this was added, the file format didn't need to change at all. So the format (and consequently, cg_annotate) could be used by other tools.

The file format:

```
file         ::= desc_line* cmd_line events_line data_line+ summary_line
desc_line    ::= "desc:" ws? non_nl_string
cmd_line     ::= "cmd:" ws? cmd
events_line  ::= "events:" ws? (event ws)+
data_line    ::= file_line | fn_line | count_line
file_line    ::= "fl=" filename
fn_line      ::= "fn=" fn_name
count_line   ::= line_num (ws+ count)* ws*
summary_line ::= "summary:" ws? count (ws+ count)+ ws*
count        ::= num
```

Where:

- `non_nl_string` is any string not containing a newline.
- `cmd` is a string holding the command line of the profiled program.
- `event` is a string containing no whitespace.
- `filename` and `fn_name` are strings.
- `num` and `line_num` are decimal numbers.
- `ws` is whitespace.

The contents of the "desc:" lines are printed out at the top of the summary. This is a generic way of providing simulation specific information, e.g. for giving the cache configuration for cache simulation.

More than one line of info can be present for each file/fn/line number. In such cases, the counts for the named events will be accumulated.

The number of counts in each `line` and the `summary_line` should not exceed the number of events in the `event_line`. If the number in each `line` is less, cg_annotate treats those missing as though they were a "0" entry. This can reduce file size.

A `file_line` changes the current file name. A `fn_line` changes the current function name. A `count_line` contains counts that pertain to the current filename/fn_name. A "fn=" `file_line` and a `fn_line` must appear before any `count_lines` to give the context of the first `count_lines`.

Similarly, each `file_line` must be immediately followed by a `fn_line`.

The summary line is redundant, because it just holds the total counts for each event. But this serves as a useful sanity check of the data; if the totals for each event don't match the summary line, something has gone wrong.

6. Callgrind: a call-graph generating cache and branch prediction profiler

To use this tool, you must specify `--tool=callgrind` on the Valgrind command line.

6.1. Overview

Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls. Optionally, cache simulation and/or branch prediction (similar to Cachegrind) can produce further information about the runtime behavior of an application.

The profile data is written out to a file at program termination. For presentation of the data, and interactive control of the profiling, two command line tools are provided:

callgrind_annotate

This command reads in the profile data, and prints a sorted lists of functions, optionally with source annotation.

For graphical visualization of the data, try KCachegrind, which is a KDE/Qt based GUI that makes it easy to navigate the large amount of data that Callgrind produces.

callgrind_control

This command enables you to interactively observe and control the status of a program currently running under Callgrind's control, without stopping the program. You can get statistics information as well as the current stack trace, and you can request zeroing of counters or dumping of profile data.

6.1.1. Functionality

Cachegrind collects flat profile data: event counts (data reads, cache misses, etc.) are attributed directly to the function they occurred in. This cost attribution mechanism is called *self* or *exclusive* attribution.

Callgrind extends this functionality by propagating costs across function call boundaries. If function `foo` calls `bar`, the costs from `bar` are added into `foo`'s costs. When applied to the program as a whole, this builds up a picture of so called *inclusive* costs, that is, where the cost of each function includes the costs of all functions it called, directly or indirectly.

As an example, the inclusive cost of `main` should be almost 100 percent of the total program cost. Because of costs arising before `main` is run, such as initialization of the run time linker and construction of global C++ objects, the inclusive cost of `main` is not exactly 100 percent of the total program cost.

Together with the call graph, this allows you to find the specific call chains starting from `main` in which the majority of the program's costs occur. Caller/callee cost attribution is also useful for profiling functions called from multiple call sites, and where optimization opportunities depend on changing code in the callers, in particular by reducing the call count.

Callgrind's cache simulation is based on that of Cachegrind. Read the documentation for Cachegrind: a cache and branch-prediction profiler first. The material below describes the features supported in addition to Cachegrind's features.

Callgrind's ability to detect function calls and returns depends on the instruction set of the platform it is run on. It works best on x86 and amd64, and unfortunately currently does not work so well on PowerPC, ARM, Thumb or MIPS code. This is because there are no explicit call or return instructions in these instruction sets, so Callgrind has to rely on heuristics to detect calls and returns.

6.1.2. Basic Usage

As with Cachegrind, you probably want to compile with debugging info (the `-g` option) and with optimization turned on.

To start a profile run for a program, execute:

```
valgrind --tool=callgrind [callgrind options] your-program [program options]
```

While the simulation is running, you can observe execution with:

```
callgrind_control -b
```

This will print out the current backtrace. To annotate the backtrace with event counts, run

```
callgrind_control -e -b
```

After program termination, a profile data file named `callgrind.out.<pid>` is generated, where *pid* is the process ID of the program being profiled. The data file contains information about the calls made in the program among the functions executed, together with **Instruction Read** (Ir) event counts.

To generate a function-by-function summary from the profile data file, use

```
callgrind_annotate [options] callgrind.out.<pid>
```

This summary is similar to the output you get from a Cachegrind run with cg_annotate: the list of functions is ordered by exclusive cost of functions, which also are the ones that are shown. Important for the additional features of Callgrind are the following two options:

- `--inclusive=yes`: Instead of using exclusive cost of functions as sorting order, use and show inclusive cost.
- `--tree=both`: Interleave into the top level list of functions, information on the callers and the callees of each function. In these lines, which represents executed calls, the cost gives the number of events spent in the call. Indented, above each function, there is the list of callers, and below, the list of callees. The sum of events in calls to a given function (caller lines), as well as the sum of events in calls from the function (callee lines) together with the self cost, gives the total inclusive cost of the function.

By default, you will also get annotated source code for all relevant functions for which the source can be found. In addition to source annotation as produced by `cg_annotate`, you will see the annotated call sites with call counts. For all other options, consult the (Cachegrind) documentation for `cg_annotate`.

For better call graph browsing experience, it is highly recommended to use KCachegrind. If your code has a significant fraction of its cost in *cycles* (sets of functions calling each other in a recursive manner), you have to use KCachegrind, as `callgrind_annotate` currently does not do any cycle detection, which is important to get correct results in this case.

If you are additionally interested in measuring the cache behavior of your program, use Callgrind with the option `--cache-sim=yes`. For branch prediction simulation, use `--branch-sim=yes`. Expect a further slow down approximately by a factor of 2.

If the program section you want to profile is somewhere in the middle of the run, it is beneficial to *fast forward* to this section without any profiling, and then enable profiling. This is achieved by using the command line option `--instr-atstart=no` and running, in a shell: `callgrind_control -i on` just before the interesting code section is executed. To exactly specify the code position where profiling should start, use the client request `CALLGRIND_START_INSTRUMENTATION`.

If you want to be able to see assembly code level annotation, specify `--dump-instr=yes`. This will produce profile data at instruction granularity. Note that the resulting profile data can only be viewed with KCachegrind. For assembly annotation, it also is interesting to see more details of the control flow inside of functions, i.e. (conditional) jumps. This will be collected by further specifying `--collect-jumps=yes`.

6.2. Advanced Usage

6.2.1. Multiple profiling dumps from one program run

Sometimes you are not interested in characteristics of a full program run, but only of a small part of it, for example execution of one algorithm. If there are multiple algorithms, or one algorithm running with different input data, it may even be useful to get different profile information for different parts of a single program run.

Profile data files have names of the form

```
callgrind.out.pid.part-threadID
```

where *pid* is the PID of the running program, *part* is a number incremented on each dump (".part" is skipped for the dump at program termination), and *threadID* is a thread identification ("-threadID" is only used if you request dumps of individual threads with `--separate-threads=yes`).

There are different ways to generate multiple profile dumps while a program is running under Callgrind's supervision. Nevertheless, all methods trigger the same action, which is "dump all profile information since the last dump or program start, and zero cost counters afterwards". To allow for zeroing cost counters without dumping, there is a second action "zero all cost counters now". The different methods are:

- **Dump on program termination.** This method is the standard way and doesn't need any special action on your part.

- **Spontaneous, interactive dumping.** Use

  ```
  callgrind_control -d [hint [PID/Name]]
  ```

  to request the dumping of profile information of the supervised application with PID or Name. *hint* is an arbitrary string you can optionally specify to later be able to distinguish profile dumps. The control program will not terminate before the dump is completely written. Note that the application must be actively running for detection of the dump command. So, for a GUI application, resize the window, or for a server, send a request.

  If you are using KCachegrind for browsing of profile information, you can use the toolbar button **Force dump**. This will request a dump and trigger a reload after the dump is written.

- **Periodic dumping after execution of a specified number of basic blocks**. For this, use the command line option `--dump-every-bb=count`.

- **Dumping at enter/leave of specified functions.** Use the option `--dump-before=function` and `--dump-after=function`. To zero cost counters before entering a function, use `--zero-before=function`.

  You can specify these options multiple times for different functions. Function specifications support wildcards: e.g. use `--dump-before='foo*'` to generate dumps before entering any function starting with *foo*.

- **Program controlled dumping.** Insert `CALLGRIND_DUMP_STATS;` at the position in your code where you want a profile dump to happen. Use `CALLGRIND_ZERO_STATS;` to only zero profile counters. See Client request reference for more information on Callgrind specific client requests.

If you are running a multi-threaded application and specify the command line option `--separate-threads=yes`, every thread will be profiled on its own and will create its own profile dump. Thus, the last two methods will only generate one dump of the currently running thread. With the other methods, you will get multiple dumps (one for each thread) on a dump request.

6.2.2. Limiting the range of collected events

By default, whenever events are happening (such as an instruction execution or cache hit/miss), Callgrind is aggregating them into event counters. However, you may be interested only in what is happening within a given function or starting from a given program phase. To this end, you can disable event aggregation for uninteresting program parts. While attribution of events to functions as well as producing separate output per program phase can be done by other means (see previous section), there are two benefits by disabling aggregation. First, this is very fine-granular (e.g. just for a loop within a function). Second, disabling event aggregation for complete program phases allows to switch off time-consuming cache simulation and allows Callgrind to progress at much higher speed with an slowdown of around factor 2 (identical to `valgrind --tool=none`).

There are two aspects which influence whether Callgrind is aggregating events at some point in time of program execution. First, there is the *collection state*. If this is off, no aggregation will be done. By changing the collection state, you can control event aggregation at a very fine granularity. However, there is not much difference in regard to execution speed of Callgrind. By default, collection is switched on, but can be disabled by different means (see below). Second, there is the *instrumentation mode* in which Callgrind is running. This mode either can be on or off. If instrumentation is off, no observation of actions in the program will be done and thus, no actions will be forwarded to the simulator which could trigger events. In the end, no events will be aggregated. The huge benefit is the much higher speed with instrumentation switched off. However, this only should be used with care and in a coarse fashion: every mode change resets the simulator state (ie. whether a memory block is cached or not) and flushes Valgrinds internal cache of instrumented code blocks, resulting in latency penalty at switching time. Also, cache simulator results directly after switching on instrumentation will be skewed due to identified cache misses which would not happen in reality (if you care about this warm-up effect, you should make sure to temporarly have collection state switched off directly after turning instrumentation mode on). However, switching instrumentation state is very useful to skip larger program phases such as an initialization phase. By default, instrumentation is switched on, but as with the collection state, can be changed by various means.

Callgrind can start with instrumentation mode switched off by specifying option `--instr-atstart=no`. Afterwards, instrumentation can be controlled in two ways: first, interactively with:

```
callgrind_control -i on
```

(and switching off again by specifying "off" instead of "on"). Second, instrumentation state can be programmatically changed with the macros `CALLGRIND_START_INSTRUMENTATION;` and `CALLGRIND_STOP_INSTRUMENTATION;`.

Similarly, the collection state at program start can be switched off by `--instr-atstart=no`. During execution, it can be controlled programmatically with the macro `CALLGRIND_TOGGLE_COLLECT;`. Further, you can limit event collection to a specific function by using `--toggle-collect=function`. This will toggle the collection state on entering and leaving the specified function. When this option is in effect, the default collection state at program start is "off". Only events happening while running inside of the given function will be collected. Recursive calls of the given function do not trigger any action. This option can be given multiple times to specify different functions of interest.

6.2.3. Counting global bus events

For access to shared data among threads in a multithreaded code, synchronization is required to avoid raced conditions. Synchronization primitives are usually implemented via atomic instructions. However, excessive use of such instructions can lead to performance issues.

To enable analysis of this problem, Callgrind optionally can count the number of atomic instructions executed. More precisely, for x86/x86_64, these are instructions using a lock prefix. For architectures supporting LL/SC, these are the number of SC instructions executed. For both, the term "global bus events" is used.

The short name of the event type used for global bus events is "Ge". To count global bus events, use `--collect-bus=yes`.

6.2.4. Avoiding cycles

Informally speaking, a cycle is a group of functions which call each other in a recursive way.

Formally speaking, a cycle is a nonempty set S of functions, such that for every pair of functions F and G in S, it is possible to call from F to G (possibly via intermediate functions) and also from G to F. Furthermore, S must be maximal -- that is, be the largest set of functions satisfying this property. For example, if a third function H is called from inside S and calls back into S, then H is also part of the cycle and should be included in S.

Recursion is quite usual in programs, and therefore, cycles sometimes appear in the call graph output of Callgrind. However, the title of this chapter should raise two questions: What is bad about cycles which makes you want to avoid them? And: How can cycles be avoided without changing program code?

Cycles are not bad in itself, but tend to make performance analysis of your code harder. This is because inclusive costs for calls inside of a cycle are meaningless. The definition of inclusive cost, i.e. self cost of a function plus inclusive cost of its callees, needs a topological order among functions. For cycles, this does not hold true: callees of a function in a cycle include the function itself. Therefore, KCachegrind does cycle detection and skips visualization of any inclusive cost for calls inside of cycles. Further, all functions in a cycle are collapsed into artificial functions called like `Cycle 1`.

Now, when a program exposes really big cycles (as is true for some GUI code, or in general code using event or callback based programming style), you lose the nice property to let you pinpoint the bottlenecks by following call chains from `main`, guided via inclusive cost. In addition, KCachegrind loses its ability to show interesting parts of the call graph, as it uses inclusive costs to cut off uninteresting areas.

Despite the meaningless of inclusive costs in cycles, the big drawback for visualization motivates the possibility to temporarily switch off cycle detection in KCachegrind, which can lead to misguiding visualization. However, often cycles appear because of unlucky superposition of independent call chains in a way that the profile result will see a cycle. Neglecting uninteresting calls with very small measured inclusive cost would break these cycles. In such cases, incorrect handling of cycles by not detecting them still gives meaningful profiling visualization.

It has to be noted that currently, **callgrind_annotate** does not do any cycle detection at all. For program executions with function recursion, it e.g. can print nonsense inclusive costs way above 100%.

After describing why cycles are bad for profiling, it is worth talking about cycle avoidance. The key insight here is that symbols in the profile data do not have to exactly match the symbols found in the program. Instead, the symbol name could encode additional information from the current execution context such as recursion level of the current function, or even some part of the call chain leading to the function. While encoding of additional information into symbols is quite capable of avoiding cycles, it has to be used carefully to not cause symbol explosion. The latter imposes large memory requirement for Callgrind with possible out-of-memory conditions, and big profile data files.

A further possibility to avoid cycles in Callgrind's profile data output is to simply leave out given functions in the call graph. Of course, this also skips any call information from and to an ignored function, and thus can break a cycle. Candidates for this typically are dispatcher functions in event driven code. The option to ignore calls to a function is `--fn-skip=function`. Aside from possibly breaking cycles, this is used in Callgrind to skip trampoline functions in the PLT sections for calls to functions in shared libraries. You can see the difference if you profile with `--skip-plt=no`. If a call is ignored, its cost events will be propagated to the enclosing function.

If you have a recursive function, you can distinguish the first 10 recursion levels by specifying `--separate-recs10=function`. Or for all functions with `--separate-recs=10`, but this will give you much bigger profile data files. In the profile data, you will see the recursion levels of "func" as the different functions with names "func", "func'2", "func'3" and so on.

If you have call chains "A > B > C" and "A > C > B" in your program, you usually get a "false" cycle "B <> C". Use `--separate-callers2=B --separate-callers2=C`, and functions "B" and "C" will be treated as different functions depending on the direct caller. Using the apostrophe for appending this "context" to the function name, you get "A > B'A > C'B" and "A > C'A > B'C", and there will be no cycle. Use `--separate-callers=2` to get a 2-caller dependency for all functions. Note that doing this will increase the size of profile data files.

6.2.5. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent. To start with empty profile counter values in the child, the client request `CALLGRIND_ZERO_STATS;` can be inserted into code to be executed by the child, directly after `fork`.

However, you will have to make sure that the output file format string (controlled by `--callgrind-out-file`) does contain `%p` (which is true by default). Otherwise, the outputs from the parent and child will overwrite each other or will be intermingled, which almost certainly is not what you want.

You will be able to control the new child independently from the parent via callgrind_control.

6.3. Callgrind Command-line Options

In the following, options are grouped into classes.

Some options allow the specification of a function/symbol name, such as `--dump-before=function`, or `--fn-skip=function`. All these options can be specified multiple times for different functions. In addition, the function specifications actually are patterns by supporting the use of wildcards '*' (zero or more arbitrary characters) and '?' (exactly one arbitrary character), similar to file name globbing in the shell. This feature is important especially for C++, as without wildcard usage, the function would have to be specified in full extent, including parameter signature.

6.3.1. Dump creation options

These options influence the name and format of the profile data files.

`--callgrind-out-file=<file>`

Write the profile data to `file` rather than to the default output file, `callgrind.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`. When multiple dumps are made, the file name is modified further; see below.

`--dump-line=<no|yes> [default: yes]`

This specifies that event counting should be performed at source line granularity. This allows source annotation for sources which are compiled with debug information (`-g`).

`--dump-instr=<no|yes> [default: no]`

This specifies that event counting should be performed at per-instruction granularity. This allows for assembly code annotation. Currently the results can only be displayed by KCachegrind.

`--compress-strings=<no|yes> [default: yes]`

This option influences the output format of the profile data. It specifies whether strings (file and function names) should be identified by numbers. This shrinks the file, but makes it more difficult for humans to read (which is not recommended in any case).

`--compress-pos=<no|yes> [default: yes]`

This option influences the output format of the profile data. It specifies whether numerical positions are always specified as absolute values or are allowed to be relative to previous numbers. This shrinks the file size.

`--combine-dumps=<no|yes> [default: no]`

When enabled, when multiple profile data parts are to be generated these parts are appended to the same output file. Not recommended.

6.3.2. Activity options

These options specify when actions relating to event counts are to be executed. For interactive control use callgrind_control.

`--dump-every-bb=<count> [default: 0, never]`

Dump profile data every `count` basic blocks. Whether a dump is needed is only checked when Valgrind's internal scheduler is run. Therefore, the minimum setting useful is about 100000. The count is a 64-bit value to make long dump periods possible.

`--dump-before=<function>`

Dump when entering `function`.

`--zero-before=<function>`

Zero all costs when entering `function`.

`--dump-after=<function>`

Dump when leaving `function`.

6.3.3. Data collection options

These options specify when events are to be aggregated into event counts. Also see Limiting range of event collection.

`--instr-atstart=<yes|no> [default: yes]`

Specify if you want Callgrind to start simulation and profiling from the beginning of the program. When set to `no`, Callgrind will not be able to collect any information, including calls, but it will have at most a slowdown of around 4, which is the minimum Valgrind overhead. Instrumentation can be interactively enabled via `callgrind_control -i on`.

Note that the resulting call graph will most probably not contain `main`, but will contain all the functions executed after instrumentation was enabled. Instrumentation can also be programmatically enabled/disabled. See the Callgrind include file `callgrind.h` for the macro you have to use in your source code.

For cache simulation, results will be less accurate when switching on instrumentation later in the program run, as the simulator starts with an empty cache at that moment. Switch on event collection later to cope with this error.

`--collect-atstart=<yes|no> [default: yes]`

Specify whether event collection is enabled at beginning of the profile run.

To only look at parts of your program, you have two possibilities:

1. Zero event counters before entering the program part you want to profile, and dump the event counters to a file after leaving that program part.

2. Switch on/off collection state as needed to only see event counters happening while inside of the program part you want to profile.

The second option can be used if the program part you want to profile is called many times. Option 1, i.e. creating a lot of dumps is not practical here.

Collection state can be toggled at entry and exit of a given function with the option `--toggle-collect`. If you use this option, collection state should be disabled at the beginning. Note that the specification of `--toggle-collect` implicitly sets `--collect-state=no`.

Collection state can be toggled also by inserting the client request `CALLGRIND_TOGGLE_COLLECT ;` at the needed code positions.

`--toggle-collect=<function>`

Toggle collection on entry/exit of `function`.

`--collect-jumps=<no|yes> [default: no]`

This specifies whether information for (conditional) jumps should be collected. As above, callgrind_annotate currently is not able to show you the data. You have to use KCachegrind to get jump arrows in the annotated code.

`--collect-systime=<no|yes|msec|usec|nsec> [default: no]`

This specifies whether information for system call times should be collected.

The value `no` indicates to record no system call information.

The other values indicate to record the number of system calls done (sysCount event) and the elapsed time (sysTime event) spent in system calls. The `--collect-systime` value gives the unit used for sysTime : milli seconds, micro seconds or nano seconds. With the value `nsec`, callgrind also records the cpu time spent during system calls (sysCpuTime).

The value `yes` is a synonym of `msec`. The value `nsec` is not supported on Darwin.

`--collect-bus=<no|yes> [default: no]`

This specifies whether the number of global bus events executed should be collected. The event type "Ge" is used for these events.

6.3.4. Cost entity separation options

These options specify how event counts should be attributed to execution contexts. For example, they specify whether the recursion level or the call chain leading to a function should be taken into account, and whether the thread ID should be considered. Also see Avoiding cycles.

`--separate-threads=<no|yes> [default: no]`

This option specifies whether profile data should be generated separately for every thread. If yes, the file names get "-threadID" appended.

`--separate-callers=<callers> [default: 0]`

Separate contexts by at most <callers> functions in the call chain. See Avoiding cycles.

`--separate-callers<number>=<function>`

Separate `number` callers for `function`. See Avoiding cycles.

`--separate-recs=<level> [default: 2]`

Separate function recursions by at most `level` levels. See Avoiding cycles.

`--separate-recs<number>=<function>`

Separate `number` recursions for `function`. See Avoiding cycles.

`--skip-plt=<no|yes> [default: yes]`

Ignore calls to/from PLT sections.

`--skip-direct-rec=<no|yes> [default: yes]`

Ignore direct recursions.

`--fn-skip=<function>`

Ignore calls to/from a given function. E.g. if you have a call chain A > B > C, and you specify function B to be ignored, you will only see A > C.

This is very convenient to skip functions handling callback behaviour. For example, with the signal/slot mechanism in the Qt graphics library, you only want to see the function emitting a signal to call the slots connected to that signal. First, determine the real call chain to see the functions needed to be skipped, then use this option.

6.3.5. Simulation options

`--cache-sim=<yes|no> [default: no]`

Specify if you want to do full cache simulation. By default, only instruction read accesses will be counted ("Ir"). With cache simulation, further event counters are enabled: Cache misses on instruction reads ("I1mr"/"ILmr"), data read accesses ("Dr") and related cache misses ("D1mr"/"DLmr"), data write accesses ("Dw") and related cache misses ("D1mw"/"DLmw"). For more information, see Cachegrind: a cache and branch-prediction profiler.

`--branch-sim=<yes|no> [default: no]`

Specify if you want to do branch prediction simulation. Further event counters are enabled: Number of executed conditional branches and related predictor misses ("Bc"/"Bcm"), executed indirect jumps and related misses of the jump address predictor ("Bi"/"Bim").

6.3.6. Cache simulation options

`--simulate-wb=<yes|no> [default: no]`

Specify whether write-back behavior should be simulated, allowing to distinguish LL caches misses with and without write backs. The cache model of Cachegrind/Callgrind does not specify write-through vs. write-back behavior, and this also is not relevant for the number of generated miss counts. However, with explicit write-back simulation it can be decided whether a miss triggers not only the loading of a new cache line, but also if a write back of a dirty cache line had to take place before. The new dirty miss events are ILdmr, DLdmr, and DLdmw, for misses because of instruction read, data read, and data write, respectively. As they produce two memory transactions, they should account for a doubled time estimation in relation to a normal miss.

`--simulate-hwpref=<yes|no> [default: no]`

Specify whether simulation of a hardware prefetcher should be added which is able to detect stream access in the second level cache by comparing accesses to separate to each page. As the simulation can not decide about any timing issues of prefetching, it is assumed that any hardware prefetch triggered succeeds before a real access is done. Thus, this gives a best-case scenario by covering all possible stream accesses.

`--cacheuse=<yes|no> [default: no]`

Specify whether cache line use should be collected. For every cache line, from loading to it being evicted, the number of accesses as well as the number of actually used bytes is determined. This behavior is related to the code which triggered loading of the cache line. In contrast to miss counters, which shows the position where the symptoms of bad cache behavior (i.e. latencies) happens, the use counters try to pinpoint at the reason (i.e. the code with the bad access behavior). The new counters are defined in a way such that worse behavior results in higher cost. AcCost1 and AcCost2 are counters showing bad temporal locality for L1 and LL caches, respectively. This is done by summing up reciprocal values of the numbers of accesses of each cache line, multiplied by 1000 (as only integer costs are allowed). E.g. for a given source line with 5 read accesses, a value of 5000 AcCost means that for every access, a new cache line was loaded and directly evicted afterwards without further accesses. Similarly, SpLoss1/2 shows bad spatial locality for L1 and LL caches, respectively. It gives the *spatial loss* count of bytes which were loaded into cache but never accessed. It pinpoints at code accessing data in a way such that cache space is wasted. This hints at bad layout of data structures in memory. Assuming a cache line size of 64 bytes and 100 L1 misses for a given source line, the loading of 6400 bytes into L1 was triggered. If SpLoss1 shows a value of 3200 for this line, this means that half of the loaded data was never used, or using a better data layout, only half of the cache space would have been needed. Please note that for cache line use counters, it currently is not possible to provide meaningful inclusive costs. Therefore, inclusive cost of these counters should be ignored.

`--I1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 instruction cache.

`--D1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 data cache.

`--LL=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the last-level cache.

6.4. Callgrind Monitor Commands

The Callgrind tool provides monitor commands handled by the Valgrind gdbserver (see Monitor command handling by the Valgrind gdbserver). Valgrind python code provides GDB front end commands giving an easier usage of the callgrind monitor commands (see GDB front end commands for Valgrind gdbserver monitor commands). To launch a callgrind monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `callgrind` command (or the shorter aliases `cg`). Using the callgrind GDB front end command provide a more flexible usage, such as auto-completion of the command by GDB. In GDB, you can use `help callgrind` to get help about the callgrind front end monitor commands and you can use `apropos callgrind` to get all the commands mentionning the word "callgrind" in their name or on-line help.

- `dump [<dump_hint>]` requests to dump the profile data.
- `zero` requests to zero the profile data counters.
- `instrumentation [on|off]` requests to set (if parameter on/off is given) or get the current instrumentation state.

- `status` requests to print out some status information.

6.5. Callgrind specific client requests

Callgrind provides the following specific client requests in `callgrind.h`. See that file for the exact details of their arguments.

`CALLGRIND_DUMP_STATS`

Force generation of a profile dump at specified position in code, for the current thread only. Written counters will be reset to zero.

`CALLGRIND_DUMP_STATS_AT(string)`

Same as `CALLGRIND_DUMP_STATS`, but allows to specify a string to be able to distinguish profile dumps.

`CALLGRIND_ZERO_STATS`

Reset the profile counters for the current thread to zero.

`CALLGRIND_TOGGLE_COLLECT`

Toggle the collection state. This allows to ignore events with regard to profile counters. See also options `--collect-atstart` and `--toggle-collect`.

`CALLGRIND_START_INSTRUMENTATION`

Start full Callgrind instrumentation if not already enabled. When cache simulation is done, this will flush the simulated cache and lead to an artificial cache warmup phase afterwards with cache misses which would not have happened in reality. See also option `--instr-atstart`.

`CALLGRIND_STOP_INSTRUMENTATION`

Stop full Callgrind instrumentation if not already disabled. This flushes Valgrinds translation cache, and does no additional instrumentation afterwards: it effectivly will run at the same speed as Nulgrind, i.e. at minimal slowdown. Use this to speed up the Callgrind run for uninteresting code parts. Use `CALLGRIND_START_INSTRUMENTATION` to enable instrumentation again. See also option `--instr-atstart`.

6.6. callgrind_annotate Command-line Options

`-h --help`

Show summary of options.

`--version`

Show version of callgrind_annotate.

`--show=A,B,C [default: all]`

Only show figures for events A,B,C.

`--threshold=<0--100> [default: 99%]`

Percentage of counts (of primary sort event) we are interested in.

callgrind_annotate stops printing functions when the sum of the cost percentage of the printed functions is bigger or equal to the given threshold percentage.

`--sort=A,B,C`

Sort columns by events A,B,C [event column order].

Optionally, each event is followed by a : and a threshold, to specify different thresholds depending on the event.

callgrind_annotate stops printing functions when the sum of the cost percentage of the printed functions for all the events is bigger or equal to the given event threshold percentages.

When one or more thresholds are given via this option, the value of `--threshold` is ignored.

`--show-percs=<no|yes> [default: no]`

When enabled, a percentage is printed next to all event counts. This helps gauge the relative importance of each function and line.

`--auto=<yes|no> [default: yes]`

Annotate all source files containing functions that helped reach the event count threshold.

`--context=N [default: 8]`

Print N lines of context before and after annotated lines.

`--inclusive=<yes|no> [default: no]`

Add subroutine costs to functions calls.

`--tree=<none|caller|calling|both> [default: none]`

Print for each function their callers, the called functions or both.

`-I, --include=<dir>`

Add `dir` to the list of directories to search for source files.

6.7. callgrind_control Command-line Options

By default, callgrind_control acts on all programs run by the current user under Callgrind. It is possible to limit the actions to specified Callgrind runs by providing a list of pids or program names as argument. The default action is to give some brief information about the applications being run under Callgrind.

`-h --help`

Show a short description, usage, and summary of options.

`--version`

Show version of callgrind_control.

`-l --long`

Show also the working directory, in addition to the brief information given by default.

`-s --stat`

Show statistics information about active Callgrind runs.

`-b --back`

Show stack/back traces of each thread in active Callgrind runs. For each active function in the stack trace, also the number of invocations since program start (or last dump) is shown. This option can be combined with -e to show inclusive cost of active functions.

`-e [A,B,...]` (default: all)

Show the current per-thread, exclusive cost values of event counters. If no explicit event names are given, figures for all event types which are collected in the given Callgrind run are shown. Otherwise, only figures for event types A, B, ... are shown. If this option is combined with -b, inclusive cost for the functions of each active stack frame is provided, too.

`--dump[=<desc>]` (default: no description)

Request the dumping of profile information. Optionally, a description can be specified which is written into the dump as part of the information giving the reason which triggered the dump action. This can be used to distinguish multiple dumps.

`-z --zero`

Zero all event counters.

`-k --kill`

Force a Callgrind run to be terminated.

`--instr=<on|off>`

Switch instrumentation mode on or off. If a Callgrind run has instrumentation disabled, no simulation is done and no events are counted. This is useful to skip uninteresting program parts, as there is much less slowdown (same as with the Valgrind tool "none"). See also the Callgrind option `--instr-atstart`.

`--vgdb-prefix=<prefix>`

Specify the vgdb prefix to use by callgrind_control. callgrind_control internally uses vgdb to find and control the active Callgrind runs. If the `--vgdb-prefix` option was used for launching valgrind, then the same option must be given to callgrind_control.

7. Helgrind: a thread error detector

To use this tool, you must specify `--tool=helgrind` on the Valgrind command line.

7.1. Overview

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.

The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Helgrind can detect three classes of errors, which are discussed in detail in the next three sections:

1. Misuses of the POSIX pthreads API.

2. Potential deadlocks arising from lock ordering problems.

3. Data races -- accessing memory without adequate locking or synchronisation.

Problems like these often result in unreproducible, timing-dependent crashes, deadlocks and other misbehaviour, and can be difficult to find by other means.

Helgrind is aware of all the pthread abstractions and tracks their effects as accurately as it can. On x86 and amd64 platforms, it understands and partially handles implicit locking arising from the use of the LOCK instruction prefix. On PowerPC/POWER and ARM platforms, it partially handles implicit locking arising from load-linked and store-conditional instruction pairs.

Helgrind works best when your application uses only the POSIX pthreads API. However, if you want to use custom threading primitives, you can describe their behaviour to Helgrind using the `ANNOTATE_*` macros defined in `helgrind.h`.

Helgrind also provides Execution Trees memory profiling using the command line option `--xtree-memory` and the monitor command `xtmemory`.

Following those is a section containing hints and tips on how to get the best out of Helgrind.

Then there is a summary of command-line options.

Finally, there is a brief summary of areas in which Helgrind could be improved.

7.2. Detected errors: Misuses of the POSIX pthreads API

Helgrind intercepts calls to many POSIX pthreads functions, and is therefore able to report on various common problems. Although these are unglamourous errors, their presence can lead to undefined program behaviour and hard-to-find bugs later on. The detected errors are:

- unlocking an invalid mutex

- unlocking a not-locked mutex
- unlocking a mutex held by a different thread
- destroying an invalid or a locked mutex
- recursively locking a non-recursive mutex
- deallocation of memory that contains a locked mutex
- passing mutex arguments to functions expecting reader-writer lock arguments, and vice versa
- when a POSIX pthread function fails with an error code that must be handled
- when a thread exits whilst still holding locked locks
- calling `pthread_cond_wait` with a not-locked mutex, an invalid mutex, or one locked by a different thread
- inconsistent bindings between condition variables and their associated mutexes
- invalid or duplicate initialisation of a pthread barrier
- initialisation of a pthread barrier on which threads are still waiting
- destruction of a pthread barrier object which was never initialised, or on which threads are still waiting
- waiting on an uninitialised pthread barrier
- for all of the pthreads functions that Helgrind intercepts, an error is reported, along with a stack trace, if the system threading library routine returns an error code, even if Helgrind itself detected no error

Checks pertaining to the validity of mutexes are generally also performed for reader-writer locks.

Signalling or broadcasting a condition variable when the associated mutex is unlocked is not strictly an error. The resulting thread scheduling may be unpredictable if the mutex is not held. The option `--check-cond-signal-mutex=yes|no` turns on checking for this situation. This kind of error is categorised as 'dubious'. The check is not turned on by default because some standard C and C++ libraries use condition signals/broadcasts with the associated mutex unlocked.

Various kinds of this-can't-possibly-happen events are also reported. These usually indicate bugs in the system threading library.

Reported errors always contain a primary stack trace indicating where the error was detected. They may also contain auxiliary stack traces giving additional information. In particular, most errors relating to mutexes will also tell you where that mutex first came to Helgrind's attention (the "`was first observed at`" part), so you have a chance of figuring out which mutex it is referring to. For example:

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
   at 0x4C2408D: pthread_mutex_unlock (hg_intercepts.c:492)
   by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
   by 0x40079B: main (tc09_bad_unlock.c:50)
  Lock at 0x7FEFFFA90 was first observed
   at 0x4C25D01: pthread_mutex_init (hg_intercepts.c:326)
   by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
   by 0x40079B: main (tc09_bad_unlock.c:50)
```

Helgrind has a way of summarising thread identities, as you see here with the text "`Thread #1`". This is so that it can speak about threads and sets of threads without overwhelming you with details. See below for more information on interpreting error messages.

7.3. Detected errors: Inconsistent Lock Orderings

In this section, and in general, to "acquire" a lock simply means to lock that lock, and to "release" a lock means to unlock it.

Helgrind monitors the order in which threads acquire locks. This allows it to detect potential deadlocks which could arise from the formation of cycles of locks. Detecting such inconsistencies is useful because, whilst actual deadlocks are fairly obvious, potential deadlocks may never be discovered during testing and could later lead to hard-to-diagnose in-service failures.

The simplest example of such a problem is as follows.

- Imagine some shared resource R, which, for whatever reason, is guarded by two locks, L1 and L2, which must both be held when R is accessed.
- Suppose a thread acquires L1, then L2, and proceeds to access R. The implication of this is that all threads in the program must acquire the two locks in the order first L1 then L2. Not doing so risks deadlock.
- The deadlock could happen if two threads -- call them T1 and T2 -- both want to access R. Suppose T1 acquires L1 first, and T2 acquires L2 first. Then T1 tries to acquire L2, and T2 tries to acquire L1, but those locks are both already held. So T1 and T2 become deadlocked.

Helgrind builds a directed graph indicating the order in which locks have been acquired in the past. When a thread acquires a new lock, the graph is updated, and then checked to see if it now contains a cycle. The presence of a cycle indicates a potential deadlock involving the locks in the cycle.

In general, Helgrind will choose two locks involved in the cycle and show you how their acquisition ordering has become inconsistent. It does this by showing the program points that first defined the ordering, and the program points which later violated it. Here is a simple example involving just two locks:

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF0006A0" violated

Observed (incorrect) order is: acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x400825: main (tc13_laog1.c:23)

 followed by a later acquisition of lock at 0x7FF0006D0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x400853: main (tc13_laog1.c:24)

Required order was established by acquisition of lock at 0x7FF0006D0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40076D: main (tc13_laog1.c:17)

 followed by a later acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40079B: main (tc13_laog1.c:18)
```

When there are more than two locks in the cycle, the error is equally serious. However, at present Helgrind does not show the locks involved, sometimes because that information is not available, but also so as to avoid flooding you with information. For example, a naive implementation of the famous Dining Philosophers problem involves a cycle of five locks (see `helgrind/tests/tc14_laog_dinphils.c`). In this case Helgrind has detected that all 5 philosophers could simultaneously pick up their left fork and then deadlock whilst waiting to pick up their right forks.

```
Thread #6: lock order "0x80499A0 before 0x8049A00" violated

Observed (incorrect) order is: acquisition of lock at 0x8049A00
   at 0x40085BC: pthread_mutex_lock (hg_intercepts.c:495)
   by 0x80485B4: dine (tc14_laog_dinphils.c:18)
   by 0x400BDA4: mythread_wrapper (hg_intercepts.c:219)
   by 0x39B924: start_thread (pthread_create.c:297)
   by 0x2F107D: clone (clone.S:130)

 followed by a later acquisition of lock at 0x80499A0
   at 0x40085BC: pthread_mutex_lock (hg_intercepts.c:495)
   by 0x80485CD: dine (tc14_laog_dinphils.c:19)
   by 0x400BDA4: mythread_wrapper (hg_intercepts.c:219)
   by 0x39B924: start_thread (pthread_create.c:297)
   by 0x2F107D: clone (clone.S:130)
```

7.4. Detected errors: Data Races

A data race happens, or could happen, when two threads access a shared memory location without using suitable locks or other synchronisation to ensure single-threaded access. Such missing locking can cause obscure timing dependent bugs. Ensuring programs are race-free is one of the central difficulties of threaded programming.

Reliably detecting races is a difficult problem, and most of Helgrind's internals are devoted to dealing with it. We begin with a simple example.

7.4.1. A Simple Data Race

About the simplest possible example of a race is as follows. In this program, it is impossible to know what the value of `var` is at the end of the program. Is it 2 ? Or 1 ?

```
#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
   var++; /* Unprotected relative to parent */ /* this is line 6 */
   return NULL;
}

int main ( void ) {
   pthread_t child;
   pthread_create(&child, NULL, child_fn, NULL);
   var++; /* Unprotected relative to child */ /* this is line 13 */
   pthread_join(child, NULL);
   return 0;
}
```

The problem is there is nothing to stop `var` being updated simultaneously by both threads. A correct program would protect `var` with a lock of type `pthread_mutex_t`, which is acquired before each access and released afterwards. Helgrind's output for this program is:

```
Thread #1 is the program's root thread

Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
   by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
   by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
   by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
   at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x4005DC: child_fn (simple_race.c:6)
   by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
   by 0x511C0CC: clone (in /lib64/libc-2.8.so)

Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```

This is quite a lot of detail for an apparently simple error. The last clause is the main error message. It says there is a race as a result of a read of size 4 (bytes), at 0x601038, which is the address of `var`, happening in function `main` at line 13 in the program.

Two important parts of the message are:

- Helgrind shows two stack traces for the error, not one. By definition, a race involves two different threads accessing the same location in such a way that the result depends on the relative speeds of the two threads.

  The first stack trace follows the text "`Possible data race during read of size 4 ...`" and the second trace follows the text "`This conflicts with a previous write of size 4 ...`". Helgrind is usually able to show both accesses involved in a race. At least one of these will be a write (since two concurrent, unsynchronised reads are harmless), and they will of course be from different threads.

  By examining your program at the two locations, you should be able to get at least some idea of what the root cause of the problem is. For each location, Helgrind shows the set of locks held at the time of the access. This often makes it clear which thread, if any, failed to take a required lock. In this example neither thread holds a lock during the access.

- For races which occur on global or stack variables, Helgrind tries to identify the name and defining point of the variable. Hence the text "`Location 0x601038 is 0 bytes inside global var "var" declared at simple_race.c:3`".

  Showing names of stack and global variables carries no run-time overhead once Helgrind has your program up and running. However, it does require Helgrind to spend considerable extra time and memory at program startup to read the relevant debug info. Hence this facility is disabled by default. To enable it, you need to give the `--read-var-info=yes` option to Helgrind.

The following section explains Helgrind's race detection algorithm in more detail.

7.4.2. Helgrind's Race Detection Algorithm

Most programmers think about threaded programming in terms of the basic functionality provided by the threading library (POSIX Pthreads): thread creation, thread joining, locks, condition variables, semaphores and barriers.

The effect of using these functions is to impose constraints upon the order in which memory accesses can happen. This implied ordering is generally known as the "happens-before relation". Once you understand the happens-before relation, it is easy to see how Helgrind finds races in your code. Fortunately, the happens-before relation is itself easy to understand, and is by itself a useful tool for reasoning about the behaviour of parallel programs. We now introduce it using a simple example.

Consider first the following buggy program:

```
Parent thread:                         Child thread:

int var;

// create child thread
pthread_create(...)
var = 20;                              var = 10;
                                       exit

// wait for child
pthread_join(...)
printf("%d\n", var);
```

The parent thread creates a child. Both then write different values to some variable `var`, and the parent then waits for the child to exit.

What is the value of `var` at the end of the program, 10 or 20? We don't know. The program is considered buggy (it has a race) because the final value of `var` depends on the relative rates of progress of the parent and child threads. If the parent is fast and the child is slow, then the child's assignment may happen later, so the final value will be 10; and vice versa if the child is faster than the parent.

The relative rates of progress of parent vs child is not something the programmer can control, and will often change from run to run. It depends on factors such as the load on the machine, what else is running, the kernel's scheduling strategy, and many other factors.

The obvious fix is to use a lock to protect `var`. It is however instructive to consider a somewhat more abstract solution, which is to send a message from one thread to the other:

```
Parent thread:                         Child thread:

int var;

// create child thread
pthread_create(...)
var = 20;
// send message to child
                                       // wait for message to arrive
                                       var = 10;
                                       exit
```

```
// wait for child
pthread_join(...)
printf("%d\n", var);
```

Now the program reliably prints "10", regardless of the speed of the threads. Why? Because the child's assignment cannot happen until after it receives the message. And the message is not sent until after the parent's assignment is done.

The message transmission creates a "happens-before" dependency between the two assignments: `var = 20;` must now happen-before `var = 10;`. And so there is no longer a race on `var`.

Note that it's not significant that the parent sends a message to the child. Sending a message from the child (after its assignment) to the parent (before its assignment) would also fix the problem, causing the program to reliably print "20".

Helgrind's algorithm is (conceptually) very simple. It monitors all accesses to memory locations. If a location -- in this example, `var`, is accessed by two different threads, Helgrind checks to see if the two accesses are ordered by the happens-before relation. If so, that's fine; if not, it reports a race.

It is important to understand that the happens-before relation creates only a partial ordering, not a total ordering. An example of a total ordering is comparison of numbers: for any two numbers `x` and `y`, either `x` is less than, equal to, or greater than `y`. A partial ordering is like a total ordering, but it can also express the concept that two elements are neither equal, less or greater, but merely unordered with respect to each other.

In the fixed example above, we say that `var = 20;` "happens-before" `var = 10;`. But in the original version, they are unordered: we cannot say that either happens-before the other.

What does it mean to say that two accesses from different threads are ordered by the happens-before relation? It means that there is some chain of inter-thread synchronisation operations which cause those accesses to happen in a particular order, irrespective of the actual rates of progress of the individual threads. This is a required property for a reliable threaded program, which is why Helgrind checks for it.

The happens-before relations created by standard threading primitives are as follows:

- When a mutex is unlocked by thread T1 and later (or immediately) locked by thread T2, then the memory accesses in T1 prior to the unlock must happen-before those in T2 after it acquires the lock.
- The same idea applies to reader-writer locks, although with some complication so as to allow correct handling of reads vs writes.
- When a condition variable (CV) is signalled on by thread T1 and some other thread T2 is thereby released from a wait on the same CV, then the memory accesses in T1 prior to the signalling must happen-before those in T2 after it returns from the wait. If no thread was waiting on the CV then there is no effect.
- If instead T1 broadcasts on a CV, then all of the waiting threads, rather than just one of them, acquire a happens-before dependency on the broadcasting thread at the point it did the broadcast.
- A thread T2 that continues after completing sem_wait on a semaphore that thread T1 posts on, acquires a happens-before dependence on the posting thread, a bit like dependencies caused mutex unlock-lock pairs. However, since a semaphore can be posted on many times, it is unspecified from which of the post calls the wait call gets its happens-before dependency.
- For a group of threads T1 .. Tn which arrive at a barrier and then move on, each thread after the call has a happens-after dependency from all threads before the barrier.
- A newly-created child thread acquires an initial happens-after dependency on the point where its parent created it. That is, all memory accesses performed by the parent prior to creating the child are regarded as happening-before all the accesses of the child.

- Similarly, when an exiting thread is reaped via a call to `pthread_join`, once the call returns, the reaping thread acquires a happens-after dependency relative to all memory accesses made by the exiting thread.

In summary: Helgrind intercepts the above listed events, and builds a directed acyclic graph represented the collective happens-before dependencies. It also monitors all memory accesses.

If a location is accessed by two different threads, but Helgrind cannot find any path through the happens-before graph from one access to the other, then it reports a race.

There are a couple of caveats:

- Helgrind doesn't check for a race in the case where both accesses are reads. That would be silly, since concurrent reads are harmless.
- Two accesses are considered to be ordered by the happens-before dependency even through arbitrarily long chains of synchronisation events. For example, if T1 accesses some location L, and then `pthread_cond_signals` T2, which later `pthread_cond_signals` T3, which then accesses L, then a suitable happens-before dependency exists between the first and second accesses, even though it involves two different inter-thread synchronisation events.

7.4.3. Interpreting Race Error Messages

Helgrind's race detection algorithm collects a lot of information, and tries to present it in a helpful way when a race is detected. Here's an example:

```
Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
   by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
   by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
   by 0x4008F2: main (tc21_pthonce.c:86)

Thread #3 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
   by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
   by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
   by 0x4008F2: main (tc21_pthonce.c:86)

Possible data race during read of size 4 at 0x601070 by thread #3
Locks held: none
   at 0x40087A: child (tc21_pthonce.c:74)
   by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
   by 0x511C0CC: clone (in /lib64/libc-2.8.so)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x400883: child (tc21_pthonce.c:74)
   by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
   by 0x511C0CC: clone (in /lib64/libc-2.8.so)
```

```
Location 0x601070 is 0 bytes inside local var "unprotected2"
declared at tc21_pthonce.c:51, in frame #0 of thread 3
```

Helgrind first announces the creation points of any threads referenced in the error message. This is so it can speak concisely about threads without repeatedly printing their creation point call stacks. Each thread is only ever announced once, the first time it appears in any Helgrind error message.

The main error message begins at the text "`Possible data race during read`". At the start is information you would expect to see -- address and size of the racing access, whether a read or a write, and the call stack at the point it was detected.

A second call stack is presented starting at the text "`This conflicts with a previous write`". This shows a previous access which also accessed the stated address, and which is believed to be racing against the access in the first call stack. Note that this second call stack is limited to a maximum of `--history-backtrace-size` entries with a default value of 8 to limit the memory usage.

Finally, Helgrind may attempt to give a description of the raced-on address in source level terms. In this example, it identifies it as a local variable, shows its name, declaration point, and in which frame (of the first call stack) it lives. Note that this information is only shown when `--read-var-info=yes` is specified on the command line. That's because reading the DWARF3 debug information in enough detail to capture variable type and location information makes Helgrind much slower at startup, and also requires considerable amounts of memory, for large programs.

Once you have your two call stacks, how do you find the root cause of the race?

The first thing to do is examine the source locations referred to by each call stack. They should both show an access to the same location, or variable.

Now figure out how how that location should have been made thread-safe:

- Perhaps the location was intended to be protected by a mutex? If so, you need to lock and unlock the mutex at both access points, even if one of the accesses is reported to be a read. Did you perhaps forget the locking at one or other of the accesses? To help you do this, Helgrind shows the set of locks held by each threads at the time they accessed the raced-on location.
- Alternatively, perhaps you intended to use a some other scheme to make it safe, such as signalling on a condition variable. In all such cases, try to find a synchronisation event (or a chain thereof) which separates the earlier-observed access (as shown in the second call stack) from the later-observed access (as shown in the first call stack). In other words, try to find evidence that the earlier access "happens-before" the later access. See the previous subsection for an explanation of the happens-before relation.

  The fact that Helgrind is reporting a race means it did not observe any happens-before relation between the two accesses. If Helgrind is working correctly, it should also be the case that you also cannot find any such relation, even on detailed inspection of the source code. Hopefully, though, your inspection of the code will show where the missing synchronisation operation(s) should have been.

7.5. Hints and Tips for Effective Use of Helgrind

Helgrind can be very helpful in finding and resolving threading-related problems. Like all sophisticated tools, it is most effective when you understand how to play to its strengths.

Helgrind will be less effective when you merely throw an existing threaded program at it and try to make sense of any reported errors. It will be more effective if you design threaded programs from the start in a way that helps Helgrind verify correctness. The same is true for finding memory errors with Memcheck, but applies more here, because thread checking is a harder problem. Consequently it is much easier to write a correct program for which Helgrind falsely reports (threading) errors than it is to write a correct program for which Memcheck falsely reports (memory) errors.

With that in mind, here are some tips, listed most important first, for getting reliable results and avoiding false errors. The first two are critical. Any violations of them will swamp you with huge numbers of false data-race errors.

1. Make sure your application, and all the libraries it uses, use the POSIX threading primitives. Helgrind needs to be able to see all events pertaining to thread creation, exit, locking and other synchronisation events. To do so it intercepts many POSIX pthreads functions.

   Do not roll your own threading primitives (mutexes, etc) from combinations of the Linux futex syscall, atomic counters, etc. These throw Helgrind's internal what's-going-on models way off course and will give bogus results.

   Also, do not reimplement existing POSIX abstractions using other POSIX abstractions. For example, don't build your own semaphore routines or reader-writer locks from POSIX mutexes and condition variables. Instead use POSIX reader-writer locks and semaphores directly, since Helgrind supports them directly.

   Helgrind directly supports the following POSIX threading abstractions: mutexes, reader-writer locks, condition variables (but see below), semaphores and barriers. Currently spinlocks are not supported, although they could be in future.

   At the time of writing, the following popular Linux packages are known to implement their own threading primitives:

   - Qt version 4.X. Qt 3.X is harmless in that it only uses POSIX pthreads primitives. Unfortunately Qt 4.X has its own implementation of mutexes (QMutex) and thread reaping. Helgrind 3.4.x contains direct support for Qt 4.X threading, which is experimental but is believed to work fairly well. A side effect of supporting Qt 4 directly is that Helgrind can be used to debug KDE4 applications. As this is an experimental feature, we would particularly appreciate feedback from folks who have used Helgrind to successfully debug Qt 4 and/or KDE4 applications.

   - Runtime support library for GNU OpenMP (part of GCC), at least for GCC versions 4.2 and 4.3. The GNU OpenMP runtime library (`libgomp.so`) constructs its own synchronisation primitives using combinations of atomic memory instructions and the futex syscall, which causes total chaos since in Helgrind since it cannot "see" those.

     Fortunately, this can be solved using a configuration-time option (for GCC). Rebuild GCC from source, and configure using `--disable-linux-futex`. This makes libgomp.so use the standard POSIX threading primitives instead. Note that this was tested using GCC 4.2.3 and has not been re-tested using more recent GCC versions. We would appreciate hearing about any successes or failures with more recent versions.

   If you must implement your own threading primitives, there are a set of client request macros in `helgrind.h` to help you describe your primitives to Helgrind. You should be able to mark up mutexes, condition variables, etc, without difficulty.

   It is also possible to mark up the effects of thread-safe reference counting using the `ANNOTATE_HAPPENS_BEFORE`, `ANNOTATE_HAPPENS_AFTER` and `ANNOTATE_HAPPENS_BEFORE_FORGET_ALL`, macros. Thread-safe reference counting using an atomically incremented/decremented refcount variable causes Helgrind problems because a one-to-zero transition of the reference count means the accessing thread has exclusive ownership of the associated resource (normally, a C++ object) and can therefore access it (normally, to run its destructor) without locking. Helgrind doesn't understand this, and markup is essential to avoid false positives.

   Here are recommended guidelines for marking up thread safe reference counting in C++. You only need to mark up your release methods -- the ones which decrement the reference count. Given a class like this:

   ```
   class MyClass {
      unsigned int mRefCount;
   ```

```
   void Release ( void ) {
      unsigned int newCount = atomic_decrement(&mRefCount);
      if (newCount == 0) {
         delete this;
      }
   }
}
```

the release method should be marked up as follows:

```
   void Release ( void ) {
      unsigned int newCount = atomic_decrement(&mRefCount);
      if (newCount == 0) {
         ANNOTATE_HAPPENS_AFTER(&mRefCount);
         ANNOTATE_HAPPENS_BEFORE_FORGET_ALL(&mRefCount);
         delete this;
      } else {
         ANNOTATE_HAPPENS_BEFORE(&mRefCount);
      }
   }
```

There are a number of complex, mostly-theoretical objections to this scheme. From a theoretical standpoint it appears to be impossible to devise a markup scheme which is completely correct in the sense of guaranteeing to remove all false races. The proposed scheme however works well in practice.

2. Avoid memory recycling. If you can't avoid it, you must use tell Helgrind what is going on via the `VALGRIND_HG_CLEAN_MEMORY` client request (in `helgrind.h`).

   Helgrind is aware of standard heap memory allocation and deallocation that occurs via `malloc`/`free`/`new`/`delete` and from entry and exit of stack frames. In particular, when memory is deallocated via `free`, `delete`, or function exit, Helgrind considers that memory clean, so when it is eventually reallocated, its history is irrelevant.

   However, it is common practice to implement memory recycling schemes. In these, memory to be freed is not handed to `free`/`delete`, but instead put into a pool of free buffers to be handed out again as required. The problem is that Helgrind has no way to know that such memory is logically no longer in use, and its history is irrelevant. Hence you must make that explicit, using the `VALGRIND_HG_CLEAN_MEMORY` client request to specify the relevant address ranges. It's easiest to put these requests into the pool manager code, and use them either when memory is returned to the pool, or is allocated from it.

3. Avoid POSIX condition variables. If you can, use POSIX semaphores (`sem_t`, `sem_post`, `sem_wait`) to do inter-thread event signalling. Semaphores with an initial value of zero are particularly useful for this.

   Helgrind only partially correctly handles POSIX condition variables. This is because Helgrind can see inter-thread dependencies between a `pthread_cond_wait` call and a `pthread_cond_signal`/`pthread_cond_broadcast` call only if the waiting thread actually gets to the rendezvous first (so that it actually calls `pthread_cond_wait`). It can't see dependencies between the threads if the signaller arrives first. In the latter case, POSIX guidelines imply that the associated boolean condition still provides an inter-thread synchronisation event, but one which is invisible to Helgrind.

   The result of Helgrind missing some inter-thread synchronisation events is to cause it to report false positives.

   The root cause of this synchronisation lossage is particularly hard to understand, so an example is helpful. It was discussed at length by Arndt Muehlenfeld ("Runtime Race Detection in Multi-Threaded Programs", Dissertation, TU Graz, Austria). The canonical POSIX-recommended usage scheme for condition variables is as follows:

```
b   is a Boolean condition, which is False most of the time
cv  is a condition variable
mx  is its associated mutex

Signaller:                             Waiter:

lock(mx)                               lock(mx)
b = True                               while (b == False)
signal(cv)                                wait(cv,mx)
unlock(mx)                             unlock(mx)
```

Assume `b` is False most of the time. If the waiter arrives at the rendezvous first, it enters its while-loop, waits for the signaller to signal, and eventually proceeds. Helgrind sees the signal, notes the dependency, and all is well.

If the signaller arrives first, `b` is set to true, and the signal disappears into nowhere. When the waiter later arrives, it does not enter its while-loop and simply carries on. But even in this case, the waiter code following the while-loop cannot execute until the signaller sets `b` to True. Hence there is still the same inter-thread dependency, but this time it is through an arbitrary in-memory condition, and Helgrind cannot see it.

By comparison, Helgrind's detection of inter-thread dependencies caused by semaphore operations is believed to be exactly correct.

As far as I know, a solution to this problem that does not require source-level annotation of condition-variable wait loops is beyond the current state of the art.

4. Make sure you are using a supported Linux distribution. At present, Helgrind only properly supports glibc-2.3 or later. This in turn means we only support glibc's NPTL threading implementation. The old LinuxThreads implementation is not supported.

5. If your application is using thread local variables, helgrind might report false positive race conditions on these variables, despite being very probably race free. On Linux, you can use `--sim-hints=deactivate-pthread-stack-cache-via-hack` to avoid such false positive error messages (see --sim-hints).

6. Round up all finished threads using `pthread_join`. Avoid detaching threads: don't create threads in the detached state, and don't call `pthread_detach` on existing threads.

   Using `pthread_join` to round up finished threads provides a clear synchronisation point that both Helgrind and programmers can see. If you don't call `pthread_join` on a thread, Helgrind has no way to know when it finishes, relative to any significant synchronisation points for other threads in the program. So it assumes that the thread lingers indefinitely and can potentially interfere indefinitely with the memory state of the program. It has every right to assume that -- after all, it might really be the case that, for scheduling reasons, the exiting thread did run very slowly in the last stages of its life.

7. Perform thread debugging (with Helgrind) and memory debugging (with Memcheck) together.

   Helgrind tracks the state of memory in detail, and memory management bugs in the application are liable to cause confusion. In extreme cases, applications which do many invalid reads and writes (particularly to freed memory) have been known to crash Helgrind. So, ideally, you should make your application Memcheck-clean before using Helgrind.

   It may be impossible to make your application Memcheck-clean unless you first remove threading bugs. In particular, it may be difficult to remove all reads and writes to freed memory in multithreaded C++ destructor sequences at program termination. So, ideally, you should make your application Helgrind-clean before using Memcheck.

Since this circularity is obviously unresolvable, at least bear in mind that Memcheck and Helgrind are to some extent complementary, and you may need to use them together.

8. POSIX requires that implementations of standard I/O (`printf`, `fprintf`, `fwrite`, `fread`, etc) are thread safe. Unfortunately GNU libc implements this by using internal locking primitives that Helgrind is unable to intercept. Consequently Helgrind generates many false race reports when you use these functions.

   Helgrind attempts to hide these errors using the standard Valgrind error-suppression mechanism. So, at least for simple test cases, you don't see any. Nevertheless, some may slip through. Just something to be aware of.

9. Helgrind's error checks do not work properly inside the system threading library itself (`libpthread.so`), and it usually observes large numbers of (false) errors in there. Valgrind's suppression system then filters these out, so you should not see them.

   If you see any race errors reported where `libpthread.so` or `ld.so` is the object associated with the innermost stack frame, please file a bug report at http://www.valgrind.org/.

7.6. Helgrind Command-line Options

The following end-user options are available:

`--free-is-write=no|yes [default: no]`

When enabled (not the default), Helgrind treats freeing of heap memory as if the memory was written immediately before the free. This exposes races where memory is referenced by one thread, and freed by another, but there is no observable synchronisation event to ensure that the reference happens before the free.

This functionality is new in Valgrind 3.7.0, and is regarded as experimental. It is not enabled by default because its interaction with custom memory allocators is not well understood at present. User feedback is welcomed.

`--track-lockorders=no|yes [default: yes]`

When enabled (the default), Helgrind performs lock order consistency checking. For some buggy programs, the large number of lock order errors reported can become annoying, particularly if you're only interested in race errors. You may therefore find it helpful to disable lock order checking.

`--history-level=none|approx|full [default: full]`

`--history-level=full` (the default) causes Helgrind collects enough information about "old" accesses that it can produce two stack traces in a race report -- both the stack trace for the current access, and the trace for the older, conflicting access. To limit memory usage, "old" accesses stack traces are limited to a maximum of `--history-backtrace-size` entries (default 8) or to `--num-callers` value if this value is smaller.

Collecting such information is expensive in both speed and memory, particularly for programs that do many inter-thread synchronisation events (locks, unlocks, etc). Without such information, it is more difficult to track down the root causes of races. Nonetheless, you may not need it in situations where you just want to check for the presence or absence of races, for example, when doing regression testing of a previously race-free program.

`--history-level=none` is the opposite extreme. It causes Helgrind not to collect any information about previous accesses. This can be dramatically faster than `--history-level=full`.

`--history-level=approx` provides a compromise between these two extremes. It causes Helgrind to show a full trace for the later access, and approximate information regarding the earlier access. This approximate information consists of two stacks, and the earlier access is guaranteed to have occurred somewhere between program points denoted by the two stacks. This is not as useful as showing the exact stack for the previous access (as `--history-level=full` does), but it is better than nothing, and it is almost as fast as `--history-level=none`.

`--history-backtrace-size=<number> [default: 8]`

When `--history-level=full` is selected, `--history-backtrace-size=number` indicates how many entries to record in "old" accesses stack traces.

`--delta-stacktrace=no|yes [default: yes on linux amd64/x86]`

This flag only has any effect at `--history-level=full`.

`--delta-stacktrace` configures the way Helgrind captures the stacktraces for the option `--history-level=full`. Such a stacktrace is typically needed each time a new piece of memory is read or written in a basic block of instructions.

`--delta-stacktrace=no` causes Helgrind to compute a full history stacktrace from the unwind info each time a stacktrace is needed.

`--delta-stacktrace=yes` indicates to Helgrind to derive a new stacktrace from the previous stacktrace, as long as there was no call instruction, no return instruction, or any other instruction changing the call stack since the previous stacktrace was captured. If no such instruction was executed, the new stacktrace can be derived from the previous stacktrace by just changing the top frame to the current program counter. This option can speed up Helgrind by 25% when using `--history-level=full`.

The following aspects have to be considered when using `--delta-stacktrace=yes` :

- In some cases (for example in a function prologue), the valgrind unwinder might not properly unwind the stack, due to some limitations and/or due to wrong unwind info. When using --delta-stacktrace=yes, the wrong stack trace captured in the function prologue will be kept till the next call or return.
- On the other hand, --delta-stacktrace=yes sometimes helps to obtain a correct stacktrace, for example when the unwind info allows a correct stacktrace to be done in the beginning of the sequence, but not later on in the instruction sequence.
- Determining which instructions are changing the callstack is partially based on platform dependent heuristics, which have to be tuned/validated specifically for the platform. Also, unwinding in a function prologue must be good enough to allow using --delta-stacktrace=yes. Currently, the option --delta-stacktrace=yes has been reasonably validated only on linux x86 32 bits and linux amd64 64 bits. For more details about how to validate --delta-stacktrace=yes, see debug option --hg-sanity-flags and the function check_cached_rcec_ok in libhb_core.c.

`--conflict-cache-size=N [default: 1000000]`

This flag only has any effect at `--history-level=full`.

Information about "old" conflicting accesses is stored in a cache of limited size, with LRU-style management. This is necessary because it isn't practical to store a stack trace for every single memory access made by the program. Historical information on not recently accessed locations is periodically discarded, to free up space in the cache.

This option controls the size of the cache, in terms of the number of different memory addresses for which conflicting access information is stored. If you find that Helgrind is showing race errors with only one stack instead of the expected two stacks, try increasing this value.

The minimum value is 10,000 and the maximum is 30,000,000 (thirty times the default value). Increasing the value by 1 increases Helgrind's memory requirement by very roughly 100 bytes, so the maximum value will easily eat up three extra gigabytes or so of memory.

`--check-stack-refs=no|yes [default: yes]`

By default Helgrind checks all data memory accesses made by your program. This flag enables you to skip checking for accesses to thread stacks (local variables). This can improve performance, but comes at the cost of missing races on stack-allocated data.

`--ignore-thread-creation=<yes|no> [default: no]`

Controls whether all activities during thread creation should be ignored. By default enabled only on Solaris. Solaris provides higher throughput, parallelism and scalability than other operating systems, at the cost of more fine-grained locking activity. This means for example that when a thread is created under glibc, just one big lock is used for all thread setup. Solaris libc uses several fine-grained locks and the creator thread resumes its activities as soon as possible, leaving for example stack and TLS setup sequence to the created thread. This situation confuses Helgrind as it assumes there is some false ordering in place between creator and created thread; and therefore many types of race conditions in the application would not be reported. To prevent such false ordering, this command line option is set to `yes` by default on Solaris. All activity (loads, stores, client requests) is therefore ignored during:

- pthread_create() call in the creator thread
- thread creation phase (stack and TLS setup) in the created thread

Also new memory allocated during thread creation is untracked, that is race reporting is suppressed there. DRD does the same thing implicitly. This is necessary because Solaris libc caches many objects and reuses them for different threads and that confuses Helgrind.

7.7. Helgrind Monitor Commands

The Helgrind tool provides monitor commands handled by Valgrind's built-in gdbserver (see Monitor command handling by the Valgrind gdbserver). Valgrind python code provides GDB front end commands giving an easier usage of the helgrind monitor commands (see GDB front end commands for Valgrind gdbserver monitor commands). To launch an helgrind monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `helgrind` command (or the shorter aliases `hg`). Using the helgrind GDB front end command provide a more flexible usage, such as evaluation of address and length arguments by GDB. In GDB, you can use `help helgrind` to get help about the helgrind front end monitor commands and you can use `apropos helgrind` to get all the commands mentionning the word "helgrind" in their name or on-line help.

- `info locks [lock_addr]` shows the list of locks and their status. If `lock_addr` is given, only shows the lock located at this address.

  In the following example, helgrind knows about one lock. This lock is located at the guest address `ga 0x8049a20`. The lock kind is `rdwr` indicating a reader-writer lock. Other possible lock kinds are `nonRec` (simple mutex, non recursive) and `mbRec` (simple mutex, possibly recursive). The lock kind is then followed by the list of threads helding the lock. In the below example, `R1:thread #6 tid 3` indicates that the helgrind thread #6 has acquired (once, as the counter following the letter R is one) the lock in read mode. The helgrind thread nr is incremented for each started thread. The presence of 'tid 3' indicates that the thread #6 is has not exited yet and is the valgrind tid 3. If a thread has terminated, then this is indicated with 'tid (exited)'.

  ```
  (gdb) monitor info locks
  Lock ga 0x8049a20 {
     kind   rdwr
   { R1:thread #6 tid 3 }
  }
  (gdb)
  ```

If you give the option `--read-var-info=yes`, then more information will be provided about the lock location, such as the global variable or the heap block that contains the lock:

```
Lock ga 0x8049a20 {
 Location 0x8049a20 is 0 bytes inside global var "s_rwlock"
 declared at rwlock_race.c:17
   kind   rdwr
 { R1:thread #3 tid 3 }
}
```

The GDB equivalent helgrind front end command `helgrind info locks [ADDR]` accept any address expression for its first ADDR argument.

- `accesshistory <addr> [<len>]` shows the access history recorded for <len> (default 1) bytes starting at <addr>. For each recorded access that overlaps with the given range, `accesshistory` shows the operation type (read or write), the address and size read or written, the helgrind thread nr/valgrind tid number that did the operation and the locks held by the thread at the time of the operation. The oldest access is shown first, the most recent access is shown last.

  In the following example, we see first a recorded write of 4 bytes by thread #7 that has modified the given 2 bytes range. The second recorded write is the most recent recorded write : thread #9 modified the same 2 bytes as part of a 4 bytes write operation. The list of locks held by each thread at the time of the write operation are also shown.

```
(gdb) monitor accesshistory 0x8049D8A 2
write of size 4 at 0x8049D88 by thread #7 tid 3
==6319== Locks held: 2, at address 0x8049D8C (and 1 that can't be shown)
==6319==    at 0x804865F: child_fn1 (locked_vs_unlocked2.c:29)
==6319==    by 0x400AE61: mythread_wrapper (hg_intercepts.c:234)
==6319==    by 0x39B924: start_thread (pthread_create.c:297)
==6319==    by 0x2F107D: clone (clone.S:130)

write of size 4 at 0x8049D88 by thread #9 tid 2
==6319== Locks held: 2, at addresses 0x8049DA4 0x8049DD4
==6319==    at 0x804877B: child_fn2 (locked_vs_unlocked2.c:45)
==6319==    by 0x400AE61: mythread_wrapper (hg_intercepts.c:234)
==6319==    by 0x39B924: start_thread (pthread_create.c:297)
==6319==    by 0x2F107D: clone (clone.S:130)
```

  The GDB equivalent helgrind front end command `helgrind accesshistory ADDR [LEN]` accept any address expression for its first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space, like in the following example:

```
(gdb) hg accesshistory &mx sizeof(mx)
read of size 4 at 0x1130A8 by thread #2 tid (exited)
==302== Locks held: none
==302==    at 0x1094AC: child8 (tc19_shadowmem.c:37)
==302==    by 0x10A0DF: steer (tc19_shadowmem.c:288)
==302==    by 0x48448A3: mythread_wrapper (hg_intercepts.c:406)
==302==    by 0x4879EA6: start_thread (pthread_create.c:477)
==302==    by 0x4990A2E: clone (clone.S:95)
```

- `xtmemory [<filename> default xtmemory.kcg.%p.%n]` requests Helgrind tool to produce an xtree heap memory report. See Execution Trees for a detailed explanation about execution trees.

7.8. Helgrind Client Requests

The following client requests are defined in `helgrind.h`. See that file for exact details of their arguments.

- `VALGRIND_HG_CLEAN_MEMORY`

  This makes Helgrind forget everything it knows about a specified memory range. This is particularly useful for memory allocators that wish to recycle memory.

- `ANNOTATE_HAPPENS_BEFORE`
- `ANNOTATE_HAPPENS_AFTER`
- `ANNOTATE_NEW_MEMORY`
- `ANNOTATE_RWLOCK_CREATE`
- `ANNOTATE_RWLOCK_DESTROY`
- `ANNOTATE_RWLOCK_ACQUIRED`
- `ANNOTATE_RWLOCK_RELEASED`

  These are used to describe to Helgrind, the behaviour of custom (non-POSIX) synchronisation primitives, which it otherwise has no way to understand. See comments in `helgrind.h` for further documentation.

7.9. A To-Do List for Helgrind

The following is a list of loose ends which should be tidied up some time.

- For lock order errors, print the complete lock cycle, rather than only doing for size-2 cycles as at present.
- The conflicting access mechanism sometimes mysteriously fails to show the conflicting access' stack, even when provided with unbounded storage for conflicting access info. This should be investigated.
- Document races caused by GCC's thread-unsafe code generation for speculative stores. In the interim see `http://gcc.gnu.org/ml/gcc/2007-10/msg00266.html` and `http://lkml.org/lkml/2007/10/24/673`.
- Don't update the lock-order graph, and don't check for errors, when a "try"-style lock operation happens (e.g. `pthread_mutex_trylock`). Such calls do not add any real restrictions to the locking order, since they can always fail to acquire the lock, resulting in the caller going off and doing Plan B (presumably it will have a Plan B). Doing such checks could generate false lock-order errors and confuse users.
- Performance can be very poor. Slowdowns on the order of 100:1 are not unusual. There is limited scope for performance improvements.

8. DRD: a thread error detector

To use this tool, you must specify `--tool=drd` on the Valgrind command line.

8.1. Overview

DRD is a Valgrind tool for detecting errors in multithreaded C and C++ programs. The tool works for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX threading primitives.

8.1.1. Multithreaded Programming Paradigms

There are two possible reasons for using multithreading in a program:

- To model concurrent activities. Assigning one thread to each activity can be a great simplification compared to multiplexing the states of multiple activities in a single thread. This is why most server software and embedded software is multithreaded.
- To use multiple CPU cores simultaneously for speeding up computations. This is why many High Performance Computing (HPC) applications are multithreaded.

Multithreaded programs can use one or more of the following programming paradigms. Which paradigm is appropriate depends e.g. on the application type. Some examples of multithreaded programming paradigms are:

- Locking. Data that is shared over threads is protected from concurrent accesses via locking. E.g. the POSIX threads library, the Qt library and the Boost.Thread library support this paradigm directly.
- Message passing. No data is shared between threads, but threads exchange data by passing messages to each other. Examples of implementations of the message passing paradigm are MPI and CORBA.
- Automatic parallelization. A compiler converts a sequential program into a multithreaded program. The original program may or may not contain parallelization hints. One example of such parallelization hints is the OpenMP standard. In this standard a set of directives are defined which tell a compiler how to parallelize a C, C++ or Fortran program. OpenMP is well suited for computational intensive applications. As an example, an open source image processing software package is using OpenMP to maximize performance on systems with multiple CPU cores. GCC supports the OpenMP standard from version 4.2.0 on.
- Software Transactional Memory (STM). Any data that is shared between threads is updated via transactions. After each transaction it is verified whether there were any conflicting transactions. If there were conflicts, the transaction is aborted, otherwise it is committed. This is a so-called optimistic approach. There is a prototype of the Intel C++ Compiler available that supports STM. Research about the addition of STM support to GCC is ongoing.

DRD supports any combination of multithreaded programming paradigms as long as the implementation of these paradigms is based on the POSIX threads primitives. DRD however does not support programs that use e.g. Linux' futexes directly. Attempts to analyze such programs with DRD will cause DRD to report many false positives.

8.1.2. POSIX Threads Programming Model

POSIX threads, also known as Pthreads, is the most widely available threading library on Unix systems.

The POSIX threads programming model is based on the following abstractions:

- A shared address space. All threads running within the same process share the same address space. All data, whether shared or not, is identified by its address.

- Regular load and store operations, which allow to read values from or to write values to the memory shared by all threads running in the same process.
- Atomic store and load-modify-store operations. While these are not mentioned in the POSIX threads standard, most microprocessors support atomic memory operations.
- Threads. Each thread represents a concurrent activity.
- Synchronization objects and operations on these synchronization objects. The following types of synchronization objects have been defined in the POSIX threads standard: mutexes, condition variables, semaphores, reader-writer synchronization objects, barriers and spinlocks.

Which source code statements generate which memory accesses depends on the *memory model* of the programming language being used. There is not yet a definitive memory model for the C and C++ languages. For a draft memory model, see also the document WG21/N2338: Concurrency memory model compiler consequences.

For more information about POSIX threads, see also the Single UNIX Specification version 3, also known as IEEE Std 1003.1.

8.1.3. Multithreaded Programming Problems

Depending on which multithreading paradigm is being used in a program, one or more of the following problems can occur:

- Data races. One or more threads access the same memory location without sufficient locking. Most but not all data races are programming errors and are the cause of subtle and hard-to-find bugs.
- Lock contention. One thread blocks the progress of one or more other threads by holding a lock too long.
- Improper use of the POSIX threads API. Most implementations of the POSIX threads API have been optimized for runtime speed. Such implementations will not complain on certain errors, e.g. when a mutex is being unlocked by another thread than the thread that obtained a lock on the mutex.
- Deadlock. A deadlock occurs when two or more threads wait for each other indefinitely.
- False sharing. If threads that run on different processor cores access different variables located in the same cache line frequently, this will slow down the involved threads a lot due to frequent exchange of cache lines.

Although the likelihood of the occurrence of data races can be reduced through a disciplined programming style, a tool for automatic detection of data races is a necessity when developing multithreaded software. DRD can detect these, as well as lock contention and improper use of the POSIX threads API.

8.1.4. Data Race Detection

The result of load and store operations performed by a multithreaded program depends on the order in which memory operations are performed. This order is determined by:

1. All memory operations performed by the same thread are performed in *program order*, that is, the order determined by the program source code and the results of previous load operations.
2. Synchronization operations determine certain ordering constraints on memory operations performed by different threads. These ordering constraints are called the *synchronization order*.

The combination of program order and synchronization order is called the *happens-before relationship*. This concept was first defined by S. Adve et al in the paper *Detecting data races on weak memory systems*, ACM SIGARCH Computer Architecture News, v.19 n.3, p.234-243, May 1991.

Two memory operations *conflict* if both operations are performed by different threads, refer to the same memory location and at least one of them is a store operation.

A multithreaded program is *data-race free* if all conflicting memory accesses are ordered by synchronization operations.

A well known way to ensure that a multithreaded program is data-race free is to ensure that a locking discipline is followed. It is e.g. possible to associate a mutex with each shared data item, and to hold a lock on the associated mutex while the shared data is accessed.

All programs that follow a locking discipline are data-race free, but not all data-race free programs follow a locking discipline. There exist multithreaded programs where access to shared data is arbitrated via condition variables, semaphores or barriers. As an example, a certain class of HPC applications consists of a sequence of computation steps separated in time by barriers, and where these barriers are the only means of synchronization. Although there are many conflicting memory accesses in such applications and although such applications do not make use mutexes, most of these applications do not contain data races.

There exist two different approaches for verifying the correctness of multithreaded programs at runtime. The approach of the so-called Eraser algorithm is to verify whether all shared memory accesses follow a consistent locking strategy. And the happens-before data race detectors verify directly whether all interthread memory accesses are ordered by synchronization operations. While the last approach is more complex to implement, and while it is more sensitive to OS scheduling, it is a general approach that works for all classes of multithreaded programs. An important advantage of happens-before data race detectors is that these do not report any false positives.

DRD is based on the happens-before algorithm.

8.2. Using DRD

8.2.1. DRD Command-line Options

The following command-line options are available for controlling the behavior of the DRD tool itself:

`--check-stack-var=<yes|no> [default: no]`

Controls whether DRD detects data races on stack variables. Verifying stack variables is disabled by default because most programs do not share stack variables over threads.

`--exclusive-threshold=<n> [default: off]`

Print an error message if any mutex or writer lock has been held longer than the time specified in milliseconds. This option enables the detection of lock contention.

`--join-list-vol=<n> [default: 10]`

Data races that occur between a statement at the end of one thread and another thread can be missed if memory access information is discarded immediately after a thread has been joined. This option allows one to specify for how many joined threads memory access information should be retained.

`--first-race-only=<yes|no> [default: no]`

Whether to report only the first data race that has been detected on a memory location or all data races that have been detected on a memory location.

`--free-is-write=<yes|no> [default: no]`

Whether to report races between accessing memory and freeing memory. Enabling this option may cause DRD to run slightly slower. Notes:

- Don't enable this option when using custom memory allocators that use the `VG_USERREQ__MALLOCLIKE_BLOCK` and `VG_USERREQ__FREELIKE_BLOCK` because that would result in false positives.
- Don't enable this option when using reference-counted objects because that will result in false positives, even when that code has been annotated properly with `ANNOTATE_HAPPENS_BEFORE` and `ANNOTATE_HAPPENS_AFTER`. See e.g. the output of the following command for an example: `valgrind --tool=drd --free-is-write=yes drd/tests/annotate_smart_pointer`.

`--report-signal-unlocked=<yes|no> [default: yes]`

Whether to report calls to `pthread_cond_signal` and `pthread_cond_broadcast` where the mutex associated with the signal through `pthread_cond_wait` or `pthread_cond_timed_wait`is not locked at the time the signal is sent. Sending a signal without holding a lock on the associated mutex is a common programming error which can cause subtle race conditions and unpredictable behavior. There exist some uncommon synchronization patterns however where it is safe to send a signal without holding a lock on the associated mutex.

`--segment-merging=<yes|no> [default: yes]`

Controls segment merging. Segment merging is an algorithm to limit memory usage of the data race detection algorithm. Disabling segment merging may improve the accuracy of the so-called 'other segments' displayed in race reports but can also trigger an out of memory error.

`--segment-merging-interval=<n> [default: 10]`

Perform segment merging only after the specified number of new segments have been created. This is an advanced configuration option that allows one to choose whether to minimize DRD's memory usage by choosing a low value or to let DRD run faster by choosing a slightly higher value. The optimal value for this parameter depends on the program being analyzed. The default value works well for most programs.

`--shared-threshold=<n> [default: off]`

Print an error message if a reader lock has been held longer than the specified time (in milliseconds). This option enables the detection of lock contention.

`--show-confl-seg=<yes|no> [default: yes]`

Show conflicting segments in race reports. Since this information can help to find the cause of a data race, this option is enabled by default. Disabling this option makes the output of DRD more compact.

`--show-stack-usage=<yes|no> [default: no]`

Print stack usage at thread exit time. When a program creates a large number of threads it becomes important to limit the amount of virtual memory allocated for thread stacks. This option makes it possible to observe how much stack memory has been used by each thread of the client program. Note: the DRD tool itself allocates some temporary data on the client thread stack. The space necessary for this temporary data must be allocated by the client program when it allocates stack memory, but is not included in stack usage reported by DRD.

`--ignore-thread-creation=<yes|no> [default: no]`

Controls whether all activities during thread creation should be ignored. By default enabled only on Solaris. Solaris provides higher throughput, parallelism and scalability than other operating systems, at the cost of more fine-grained locking activity. This means for example that when a thread is created under glibc, just one big lock is used for all thread setup. Solaris libc uses several fine-grained locks and the creator thread resumes its activities as soon as possible, leaving for example stack and TLS setup sequence to the created thread. This situation confuses DRD as it assumes there is some false ordering in place between creator and created thread; and therefore many types of race conditions in the application would not be reported. To prevent such false ordering, this command line option is set to yes by default on Solaris. All activity (loads, stores, client requests) is therefore ignored during:

- pthread_create() call in the creator thread
- thread creation phase (stack and TLS setup) in the created thread

The following options are available for monitoring the behavior of the client program:

--trace-addr=<address> [default: none]

Trace all load and store activity for the specified address. This option may be specified more than once.

--ptrace-addr=<address> [default: none]

Trace all load and store activity for the specified address and keep doing that even after the memory at that address has been freed and reallocated.

--trace-alloc=<yes|no> [default: no]

Trace all memory allocations and deallocations. May produce a huge amount of output.

--trace-barrier=<yes|no> [default: no]

Trace all barrier activity.

--trace-cond=<yes|no> [default: no]

Trace all condition variable activity.

--trace-fork-join=<yes|no> [default: no]

Trace all thread creation and all thread termination events.

--trace-hb=<yes|no> [default: no]

Trace execution of the ANNOTATE_HAPPENS_BEFORE(), ANNOTATE_HAPPENS_AFTER() and ANNOTATE_HAPPENS_DONE() client requests.

--trace-mutex=<yes|no> [default: no]

Trace all mutex activity.

--trace-rwlock=<yes|no> [default: no]

Trace all reader-writer lock activity.

--trace-semaphore=<yes|no> [default: no]

Trace all semaphore activity.

8.2.2. Detected Errors: Data Races

DRD prints a message every time it detects a data race. Please keep the following in mind when interpreting DRD's output:

- Every thread is assigned a *thread ID* by the DRD tool. A thread ID is a number. Thread ID's start at one and are never recycled.
- The term *segment* refers to a consecutive sequence of load, store and synchronization operations, all issued by the same thread. A segment always starts and ends at a synchronization operation. Data race analysis is performed between segments instead of between individual load and store operations because of performance reasons.
- There are always at least two memory accesses involved in a data race. Memory accesses involved in a data race are called *conflicting memory accesses*. DRD prints a report for each memory access that conflicts with a past memory access.

Below you can find an example of a message printed by DRD when it detects a data race:

```
$ valgrind --tool=drd --read-var-info=yes drd/tests/rwlock_race
...
==9466== Thread 3:
==9466== Conflicting load by thread 3 at 0x006020b8 size 4
==9466==    at 0x400B6C: thread_func (rwlock_race.c:29)
==9466==    by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==    by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==    by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Location 0x6020b8 is 0 bytes inside local var "s_racy"
==9466== declared at rwlock_race.c:18, in frame #0 of thread 3
==9466== Other segment start (thread 2)
==9466==    at 0x4C2847D: pthread_rwlock_rdlock* (drd_pthread_intercepts.c:813)
==9466==    by 0x400B6B: thread_func (rwlock_race.c:28)
==9466==    by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==    by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==    by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Other segment end (thread 2)
==9466==    at 0x4C28B54: pthread_rwlock_unlock* (drd_pthread_intercepts.c:912)
==9466==    by 0x400B84: thread_func (rwlock_race.c:30)
==9466==    by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==    by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==    by 0x53250CC: clone (in /lib64/libc-2.8.so)
...
```

The above report has the following meaning:

- The number in the column on the left is the process ID of the process being analyzed by DRD.
- The first line ("Thread 3") tells you the thread ID for the thread in which context the data race has been detected.
- The next line tells which kind of operation was performed (load or store) and by which thread. On the same line the start address and the number of bytes involved in the conflicting access are also displayed.
- Next, the call stack of the conflicting access is displayed. If your program has been compiled with debug information (`-g`), this call stack will include file names and line numbers. The two bottommost frames in this call stack (`clone` and `start_thread`) show how the NPTL starts a thread. The third frame (`vg_thread_wrapper`) is added by DRD. The fourth frame (`thread_func`) is the first interesting line because it shows the thread entry point, that is the function that has been passed as the third argument to `pthread_create`.
- Next, the allocation context for the conflicting address is displayed. For dynamically allocated data the allocation call stack is shown. For static variables and stack variables the allocation context is only shown when the option `--read-var-info=yes` has been specified. Otherwise DRD will print `Allocation context: unknown`.

- A conflicting access involves at least two memory accesses. For one of these accesses an exact call stack is displayed, and for the other accesses an approximate call stack is displayed, namely the start and the end of the segments of the other accesses. This information can be interpreted as follows:

  1. Start at the bottom of both call stacks, and count the number stack frames with identical function name, file name and line number. In the above example the three bottommost frames are identical (`clone`, `start_thread` and `vg_thread_wrapper`).

  2. The next higher stack frame in both call stacks now tells you between in which source code region the other memory access happened. The above output tells that the other memory access involved in the data race happened between source code lines 28 and 30 in file `rwlock_race.c`.

8.2.3. Detected Errors: Lock Contention

Threads must be able to make progress without being blocked for too long by other threads. Sometimes a thread has to wait until a mutex or reader-writer synchronization object is unlocked by another thread. This is called *lock contention*.

Lock contention causes delays. Such delays should be as short as possible. The two command line options `--exclusive-threshold=<n>` and `--shared-threshold=<n>` make it possible to detect excessive lock contention by making DRD report any lock that has been held longer than the specified threshold. An example:

```
$ valgrind --tool=drd --exclusive-threshold=10 drd/tests/hold_lock -i 500
...
==10668== Acquired at:
==10668==    at 0x4C267C8: pthread_mutex_lock (drd_pthread_intercepts.c:395)
==10668==    by 0x400D92: main (hold_lock.c:51)
==10668== Lock on mutex 0x7fefffd50 was held during 503 ms (threshold: 10 ms).
==10668==    at 0x4C26ADA: pthread_mutex_unlock (drd_pthread_intercepts.c:441)
==10668==    by 0x400DB5: main (hold_lock.c:55)
...
```

The `hold_lock` test program holds a lock as long as specified by the `-i` (interval) argument. The DRD output reports that the lock acquired at line 51 in source file `hold_lock.c` and released at line 55 was held during 503 ms, while a threshold of 10 ms was specified to DRD.

8.2.4. Detected Errors: Misuse of the POSIX threads API

DRD is able to detect and report the following misuses of the POSIX threads API:

- Passing the address of one type of synchronization object (e.g. a mutex) to a POSIX API call that expects a pointer to another type of synchronization object (e.g. a condition variable).
- Attempts to unlock a mutex that has not been locked.
- Attempts to unlock a mutex that was locked by another thread.
- Attempts to lock a mutex of type `PTHREAD_MUTEX_NORMAL` or a spinlock recursively.
- Destruction or deallocation of a locked mutex.
- Sending a signal to a condition variable while no lock is held on the mutex associated with the condition variable.
- Calling `pthread_cond_wait` on a mutex that is not locked, that is locked by another thread or that has been locked recursively.

- Associating two different mutexes with a condition variable through `pthread_cond_wait`.
- Destruction or deallocation of a condition variable that is being waited upon.
- Destruction or deallocation of a locked reader-writer synchronization object.
- Attempts to unlock a reader-writer synchronization object that was not locked by the calling thread.
- Attempts to recursively lock a reader-writer synchronization object exclusively.
- Attempts to pass the address of a user-defined reader-writer synchronization object to a POSIX threads function.
- Attempts to pass the address of a POSIX reader-writer synchronization object to one of the annotations for user-defined reader-writer synchronization objects.
- Reinitialization of a mutex, condition variable, reader-writer lock, semaphore or barrier.
- Destruction or deallocation of a semaphore or barrier that is being waited upon.
- Missing synchronization between barrier wait and barrier destruction.
- Exiting a thread without first unlocking the spinlocks, mutexes or reader-writer synchronization objects that were locked by that thread.
- Passing an invalid thread ID to `pthread_join` or `pthread_cancel`.

8.2.5. Client Requests

Just as for other Valgrind tools it is possible to let a client program interact with the DRD tool through client requests. In addition to the client requests several macros have been defined that allow to use the client requests in a convenient way.

The interface between client programs and the DRD tool is defined in the header file `<valgrind/drd.h>`. The available macros and client requests are:

- The macro `DRD_GET_VALGRIND_THREADID` and the corresponding client request `VG_USERREQ__DRD_GET_VALGRIND_THREAD_ID`. Query the thread ID that has been assigned by the Valgrind core to the thread executing this client request. Valgrind's thread ID's start at one and are recycled in case a thread stops.
- The macro `DRD_GET_DRD_THREADID` and the corresponding client request `VG_USERREQ__DRD_GET_DRD_THREAD_ID`. Query the thread ID that has been assigned by DRD to the thread executing this client request. These are the thread ID's reported by DRD in data race reports and in trace messages. DRD's thread ID's start at one and are never recycled.
- The macros `DRD_IGNORE_VAR(x)`, `ANNOTATE_TRACE_MEMORY(&x)` and the corresponding client request `VG_USERREQ__DRD_START_SUPPRESSION`. Some applications contain intentional races. There exist e.g. applications where the same value is assigned to a shared variable from two different threads. It may be more convenient to suppress such races than to solve these. This client request allows one to suppress such races.
- The macro `DRD_STOP_IGNORING_VAR(x)` and the corresponding client request `VG_USERREQ__DRD_FINISH_SUPPRESSION`. Tell DRD to no longer ignore data races for the address range that was suppressed either via the macro `DRD_IGNORE_VAR(x)` or via the client request `VG_USERREQ__DRD_START_SUPPRESSION`.
- The macro `DRD_TRACE_VAR(x)`. Trace all load and store activity for the address range starting at `&x` and occupying `sizeof(x)` bytes. When DRD reports a data race on a specified variable, and it's not immediately clear

which source code statements triggered the conflicting accesses, it can be very helpful to trace all activity on the offending memory location.

- The macro `DRD_STOP_TRACING_VAR(x)`. Stop tracing load and store activity for the address range starting at `&x` and occupying `sizeof(x)` bytes.
- The macro `ANNOTATE_TRACE_MEMORY(&x)`. Trace all load and store activity that touches at least the single byte at the address `&x`.
- The client request `VG_USERREQ__DRD_START_TRACE_ADDR`, which allows one to trace all load and store activity for the specified address range.
- The client request `VG_USERREQ__DRD_STOP_TRACE_ADDR`. Do no longer trace load and store activity for the specified address range.
- The macro `ANNOTATE_HAPPENS_BEFORE(addr)` tells DRD to insert a mark. Insert this macro just after an access to the variable at the specified address has been performed.
- The macro `ANNOTATE_HAPPENS_AFTER(addr)` tells DRD that the next access to the variable at the specified address should be considered to have happened after the access just before the latest `ANNOTATE_HAPPENS_BEFORE(addr)` annotation that references the same variable. The purpose of these two macros is to tell DRD about the order of inter-thread memory accesses implemented via atomic memory operations. See also `drd/tests/annotate_smart_pointer.cpp` for an example.
- The macro `ANNOTATE_RWLOCK_CREATE(rwlock)` tells DRD that the object at address `rwlock` is a reader-writer synchronization object that is not a `pthread_rwlock_t` synchronization object. See also `drd/tests/annotate_rwlock.c` for an example.
- The macro `ANNOTATE_RWLOCK_DESTROY(rwlock)` tells DRD that the reader-writer synchronization object at address `rwlock` has been destroyed.
- The macro `ANNOTATE_WRITERLOCK_ACQUIRED(rwlock)` tells DRD that a writer lock has been acquired on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_READERLOCK_ACQUIRED(rwlock)` tells DRD that a reader lock has been acquired on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_RWLOCK_ACQUIRED(rwlock, is_w)` tells DRD that a writer lock (when `is_w != 0`) or that a reader lock (when `is_w == 0`) has been acquired on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_WRITERLOCK_RELEASED(rwlock)` tells DRD that a writer lock has been released on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_READERLOCK_RELEASED(rwlock)` tells DRD that a reader lock has been released on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_RWLOCK_RELEASED(rwlock, is_w)` tells DRD that a writer lock (when `is_w != 0`) or that a reader lock (when `is_w == 0`) has been released on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_BARRIER_INIT(barrier, count, reinitialization_allowed)` tells DRD that a new barrier object at the address `barrier` has been initialized, that `count` threads participate in each barrier and also whether or not barrier reinitialization without intervening destruction should be reported as an error. See also `drd/tests/annotate_barrier.c` for an example.
- The macro `ANNOTATE_BARRIER_DESTROY(barrier)` tells DRD that a barrier object is about to be destroyed.

- The macro ANNOTATE_BARRIER_WAIT_BEFORE(barrier) tells DRD that waiting for a barrier will start.
- The macro ANNOTATE_BARRIER_WAIT_AFTER(barrier) tells DRD that waiting for a barrier has finished.
- The macro ANNOTATE_BENIGN_RACE_SIZED(addr, size, descr) tells DRD that any races detected on the specified address are benign and hence should not be reported. The descr argument is ignored but can be used to document why data races on addr are benign.
- The macro ANNOTATE_BENIGN_RACE_STATIC(var, descr) tells DRD that any races detected on the specified static variable are benign and hence should not be reported. The descr argument is ignored but can be used to document why data races on var are benign. Note: this macro can only be used in C++ programs and not in C programs.
- The macro ANNOTATE_IGNORE_READS_BEGIN tells DRD to ignore all memory loads performed by the current thread.
- The macro ANNOTATE_IGNORE_READS_END tells DRD to stop ignoring the memory loads performed by the current thread.
- The macro ANNOTATE_IGNORE_WRITES_BEGIN tells DRD to ignore all memory stores performed by the current thread.
- The macro ANNOTATE_IGNORE_WRITES_END tells DRD to stop ignoring the memory stores performed by the current thread.
- The macro ANNOTATE_IGNORE_READS_AND_WRITES_BEGIN tells DRD to ignore all memory accesses performed by the current thread.
- The macro ANNOTATE_IGNORE_READS_AND_WRITES_END tells DRD to stop ignoring the memory accesses performed by the current thread.
- The macro ANNOTATE_NEW_MEMORY(addr, size) tells DRD that the specified memory range has been allocated by a custom memory allocator in the client program and that the client program will start using this memory range.
- The macro ANNOTATE_THREAD_NAME(name) tells DRD to associate the specified name with the current thread and to include this name in the error messages printed by DRD.
- The macros VALGRIND_MALLOCLIKE_BLOCK and VALGRIND_FREELIKE_BLOCK from the Valgrind core are implemented; they are described in The Client Request mechanism.

Note: if you compiled Valgrind yourself, the header file <valgrind/drd.h> will have been installed in the directory /usr/include by the command make install. If you obtained Valgrind by installing it as a package however, you will probably have to install another package with a name like valgrind-devel before Valgrind's header files are available.

8.2.6. Debugging C++11 Programs

If you want to use the C++11 class std::thread you will need to do the following to annotate the std::shared_ptr<> objects used in the implementation of that class:

- Add the following code at the start of a common header or at the start of each source file, before any C++ header files are included:

```
#include <valgrind/drd.h>
```

```
#define _GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(addr) ANNOTATE_HAPPENS_BEFORE(addr)
#define _GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(addr)  ANNOTATE_HAPPENS_AFTER(addr)
```

- Download the gcc source code and from source file libstdc++-v3/src/c++11/thread.cc copy the implementation of the `execute_native_thread_routine()` and `std::thread::_M_start_thread()` functions into a source file that is linked with your application. Make sure that also in this source file the _GLIBCXX_SYNCHRONIZATION_HAPPENS_*() macros are defined properly.

For more information, see also *The GNU C++ Library Manual, Debugging Support* (http://gcc.gnu.org/onlinedocs/libstdc++/manual/debug.html).

8.2.7. Debugging GNOME Programs

GNOME applications use the threading primitives provided by the `glib` and `gthread` libraries. These libraries are built on top of POSIX threads, and hence are directly supported by DRD. Please keep in mind that you have to call `g_thread_init` before creating any threads, or DRD will report several data races on glib functions. See also the GLib Reference Manual for more information about `g_thread_init`.

One of the many facilities provided by the `glib` library is a block allocator, called `g_slice`. You have to disable this block allocator when using DRD by adding the following to the shell environment variables: `G_SLICE=always-malloc`. See also the GLib Reference Manual for more information.

8.2.8. Debugging Boost.Thread Programs

The Boost.Thread library is the threading library included with the cross-platform Boost Libraries. This threading library is an early implementation of the upcoming C++0x threading library.

Applications that use the Boost.Thread library should run fine under DRD.

More information about Boost.Thread can be found here:

- Anthony Williams, Boost.Thread Library Documentation, Boost website, 2007.
- Anthony Williams, What's New in Boost Threads?, Recent changes to the Boost Thread library, Dr. Dobbs Magazine, October 2008.

8.2.9. Debugging OpenMP Programs

OpenMP stands for *Open Multi-Processing*. The OpenMP standard consists of a set of compiler directives for C, C++ and Fortran programs that allows a compiler to transform a sequential program into a parallel program. OpenMP is well suited for HPC applications and allows one to work at a higher level compared to direct use of the POSIX threads API. While OpenMP ensures that the POSIX API is used correctly, OpenMP programs can still contain data races. So it definitely makes sense to verify OpenMP programs with a thread checking tool.

DRD supports OpenMP shared-memory programs generated by GCC. GCC supports OpenMP since version 4.2.0. GCC's runtime support for OpenMP programs is provided by a library called `libgomp`. The synchronization primitives implemented in this library use Linux' futex system call directly, unless the library has been configured with the `--disable-linux-futex` option. DRD only supports libgomp libraries that have been configured with this option and in which symbol information is present. For most Linux distributions this means that you will have to recompile GCC. See also the script `drd/scripts/download-and-build-gcc` in the Valgrind source tree for an example of how to compile GCC. You will also have to make sure that the newly compiled `libgomp.so` library is loaded when OpenMP programs are started. This is possible by adding a line similar to the following to your shell startup script:

```
export LD_LIBRARY_PATH=~/gcc-4.4.0/lib64:~/gcc-4.4.0/lib:
```

As an example, the test OpenMP test program `drd/tests/omp_matinv` triggers a data race when the option -r has been specified on the command line. The data race is triggered by the following code:

```
#pragma omp parallel for private(j)
for (j = 0; j < rows; j++)
{
  if (i != j)
  {
    const elem_t factor = a[j * cols + i];
    for (k = 0; k < cols; k++)
    {
      a[j * cols + k] -= a[i * cols + k] * factor;
    }
  }
}
```

The above code is racy because the variable `k` has not been declared private. DRD will print the following error message for the above code:

```
$ valgrind --tool=drd --check-stack-var=yes --read-var-info=yes drd/tests/omp_matinv 3 -t
...
Conflicting store by thread 1/1 at 0x7fefffbc4 size 4
   at 0x4014A0: gj.omp_fn.0 (omp_matinv.c:203)
   by 0x401211: gj (omp_matinv.c:159)
   by 0x40166A: invert_matrix (omp_matinv.c:238)
   by 0x4019B4: main (omp_matinv.c:316)
Location 0x7fefffbc4 is 0 bytes inside local var "k"
declared at omp_matinv.c:160, in frame #0 of thread 1
...
```

In the above output the function name `gj.omp_fn.0` has been generated by GCC from the function name `gj`. The allocation context information shows that the data race has been caused by modifying the variable `k`.

Note: for GCC versions before 4.4.0, no allocation context information is shown. With these GCC versions the most usable information in the above output is the source file name and the line number where the data race has been detected (`omp_matinv.c:203`).

For more information about OpenMP, see also openmp.org.

8.2.10. DRD and Custom Memory Allocators

DRD tracks all memory allocation events that happen via the standard memory allocation and deallocation functions (`malloc`, `free`, `new` and `delete`), via entry and exit of stack frames or that have been annotated with Valgrind's memory pool client requests. DRD uses memory allocation and deallocation information for two purposes:

- To know where the scope ends of POSIX objects that have not been destroyed explicitly. It is e.g. not required by the POSIX threads standard to call `pthread_mutex_destroy` before freeing the memory in which a mutex object resides.
- To know where the scope of variables ends. If e.g. heap memory has been used by one thread, that thread frees that memory, and another thread allocates and starts using that memory, no data races must be reported for that memory.

It is essential for correct operation of DRD that the tool knows about memory allocation and deallocation events. When analyzing a client program with DRD that uses a custom memory allocator, either instrument the custom memory allocator with the `VALGRIND_MALLOCLIKE_BLOCK` and `VALGRIND_FREELIKE_BLOCK` macros or disable the custom memory allocator.

As an example, the GNU libstdc++ library can be configured to use standard memory allocation functions instead of memory pools by setting the environment variable `GLIBCXX_FORCE_NEW`. For more information, see also the libstdc++ manual.

8.2.11. DRD Versus Memcheck

It is essential for correct operation of DRD that there are no memory errors such as dangling pointers in the client program. Which means that it is a good idea to make sure that your program is Memcheck-clean before you analyze it with DRD. It is possible however that some of the Memcheck reports are caused by data races. In this case it makes sense to run DRD before Memcheck.

So which tool should be run first? In case both DRD and Memcheck complain about a program, a possible approach is to run both tools alternatingly and to fix as many errors as possible after each run of each tool until none of the two tools prints any more error messages.

8.2.12. Resource Requirements

The requirements of DRD with regard to heap and stack memory and the effect on the execution time of client programs are as follows:

- When running a program under DRD with default DRD options, between 1.1 and 3.6 times more memory will be needed compared to a native run of the client program. More memory will be needed if loading debug information has been enabled (`--read-var-info=yes`).
- DRD allocates some of its temporary data structures on the stack of the client program threads. This amount of data is limited to 1 - 2 KB. Make sure that thread stacks are sufficiently large.
- Most applications will run between 20 and 50 times slower under DRD than a native single-threaded run. The slowdown will be most noticeable for applications which perform frequent mutex lock / unlock operations.

8.2.13. Hints and Tips for Effective Use of DRD

The following information may be helpful when using DRD:

- Make sure that debug information is present in the executable being analyzed, such that DRD can print function name and line number information in stack traces. Most compilers can be told to include debug information via compiler option `-g`.
- Compile with option `-O1` instead of `-O0`. This will reduce the amount of generated code, may reduce the amount of debug info and will speed up DRD's processing of the client program. For more information, see also Getting started.
- If DRD reports any errors on libraries that are part of your Linux distribution like e.g. `libc.so` or `libstdc++.so`, installing the debug packages for these libraries will make the output of DRD a lot more detailed.
- When using C++, do not send output from more than one thread to `std::cout`. Doing so would not only generate multiple data race reports, it could also result in output from several threads getting mixed up. Either use `printf` or do the following:

  1. Derive a class from `std::ostreambuf` and let that class send output line by line to `stdout`. This will avoid that individual lines of text produced by different threads get mixed up.

2. Create one instance of `std::ostream` for each thread. This makes stream formatting settings thread-local. Pass a per-thread instance of the class derived from `std::ostreambuf` to the constructor of each instance.

3. Let each thread send its output to its own instance of `std::ostream` instead of `std::cout`.

8.3. Using the POSIX Threads API Effectively

8.3.1. Mutex types

The Single UNIX Specification version two defines the following four mutex types (see also the documentation of `pthread_mutexattr_settype`):

- *normal*, which means that no error checking is performed, and that the mutex is non-recursive.
- *error checking*, which means that the mutex is non-recursive and that error checking is performed.
- *recursive*, which means that a mutex may be locked recursively.
- *default*, which means that error checking behavior is undefined, and that the behavior for recursive locking is also undefined. Or: portable code must neither trigger error conditions through the Pthreads API nor attempt to lock a mutex of default type recursively.

In complex applications it is not always clear from beforehand which mutex will be locked recursively and which mutex will not be locked recursively. Attempts lock a non-recursive mutex recursively will result in race conditions that are very hard to find without a thread checking tool. So either use the error checking mutex type and consistently check the return value of Pthread API mutex calls, or use the recursive mutex type.

8.3.2. Condition variables

A condition variable allows one thread to wake up one or more other threads. Condition variables are often used to notify one or more threads about state changes of shared data. Unfortunately it is very easy to introduce race conditions by using condition variables as the only means of state information propagation. A better approach is to let threads poll for changes of a state variable that is protected by a mutex, and to use condition variables only as a thread wakeup mechanism. See also the source file `drd/tests/monitor_example.cpp` for an example of how to implement this concept in C++. The monitor concept used in this example is a well known and very useful concept -- see also Wikipedia for more information about the monitor concept.

8.3.3. pthread_cond_timedwait and timeouts

Historically the function `pthread_cond_timedwait` only allowed the specification of an absolute timeout, that is a timeout independent of the time when this function was called. However, almost every call to this function expresses a relative timeout. This typically happens by passing the sum of `clock_gettime(CLOCK_REALTIME)` and a relative timeout as the third argument. This approach is incorrect since forward or backward clock adjustments by e.g. ntpd will affect the timeout. A more reliable approach is as follows:

- When initializing a condition variable through `pthread_cond_init`, specify that the timeout of `pthread_cond_timedwait` will use the clock `CLOCK_MONOTONIC` instead of `CLOCK_REALTIME`. You can do this via `pthread_condattr_setclock(..., CLOCK_MONOTONIC)`.
- When calling `pthread_cond_timedwait`, pass the sum of `clock_gettime(CLOCK_MONOTONIC)` and a relative timeout as the third argument.

See also `drd/tests/monitor_example.cpp` for an example.

8.4. Limitations

DRD currently has the following limitations:

- DRD, just like Memcheck, will refuse to start on Linux distributions where all symbol information has been removed from `ld.so`. This is e.g. the case for the PPC editions of openSUSE and Gentoo. You will have to install the glibc debuginfo package on these platforms before you can use DRD. See also openSUSE bug 396197 and Gentoo bug 214065.
- With gcc 4.4.3 and before, DRD may report data races on the C++ class `std::string` in a multithreaded program. This is a know `libstdc++` issue -- see also GCC bug 40518 for more information.
- If you compile the DRD source code yourself, you need GCC 3.0 or later. GCC 2.95 is not supported.
- Of the two POSIX threads implementations for Linux, only the NPTL (Native POSIX Thread Library) is supported. The older LinuxThreads library is not supported.

8.5. Feedback

If you have any comments, suggestions, feedback or bug reports about DRD, feel free to either post a message on the Valgrind users mailing list or to file a bug report. See also http://www.valgrind.org/ for more information.

9. Massif: a heap profiler

To use this tool, you must specify `--tool=massif` on the Valgrind command line.

9.1. Overview

Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s), although it does not do so by default.

Heap profiling can help you reduce the amount of memory your program uses. On modern machines with virtual memory, this provides the following benefits:

- It can speed up your program -- a smaller program will interact better with your machine's caches and avoid paging.
- If your program uses lots of memory, it will reduce the chance that it exhausts your machine's swap space.

Also, there are certain space leaks that aren't detected by traditional leak-checkers, such as Memcheck's. That's because the memory isn't ever actually lost -- a pointer remains to it -- but it's not in use. Programs that have leaks like this can unnecessarily increase the amount of memory they are using over time. Massif can help identify these leaks.

Importantly, Massif tells you not only how much heap memory your program is using, it also gives very detailed information that indicates which parts of your program are responsible for allocating the heap memory.

Massif also provides Execution Trees memory profiling using the command line option `--xtree-memory` and the monitor command `xtmemory`.

9.2. Using Massif and ms_print

First off, as for the other Valgrind tools, you should compile with debugging info (the `-g` option). It shouldn't matter much what optimisation level you compile your program with, as this is unlikely to affect the heap memory usage.

Then, you need to run Massif itself to gather the profiling information, and then run ms_print to present it in a readable way.

9.2.1. An Example Program

An example will make things clear. Consider the following C program (annotated with line numbers) which allocates a number of different blocks on the heap.

```
#include <stdlib.h>

void g(void)
{
   malloc(4000);
}

void f(void)
{
   malloc(2000);
   g();
```

```
}

int main(void)
{
   int i;
   int* a[10];

   for (i = 0; i < 10; i++) {
      a[i] = malloc(1000);
   }

   f();

   g();

   for (i = 0; i < 10; i++) {
      free(a[i]);
   }

   return 0;
}
```

9.2.2. Running Massif

To gather heap profiling information about the program `prog`, type:

```
valgrind --tool=massif prog
```

The program will execute (slowly). Upon completion, no summary statistics are printed to Valgrind's commentary; all of Massif's profiling data is written to a file. By default, this file is called `massif.out.<pid>`, where `<pid>` is the process ID, although this filename can be changed with the `--massif-out-file` option.

9.2.3. Running ms_print

To see the information gathered by Massif in an easy-to-read form, use ms_print. If the output file's name is `massif.out.12345`, type:

```
ms_print massif.out.12345
```

ms_print will produce (a) a graph showing the memory consumption over the program's execution, and (b) detailed information about the responsible allocation sites at various points in the program, including the point of peak memory allocation. The use of a separate script for presenting the results is deliberate: it separates the data gathering from its presentation, and means that new methods of presenting the data can be added in the future.

9.2.4. The Output Preamble

After running this program under Massif, the first part of ms_print's output contains a preamble which just states how the program, Massif and ms_print were each invoked:

```
--------------------------------------------------------------------------------
Command:            example
```

```
Massif arguments:   (none)
ms_print arguments: massif.out.12797
--------------------------------------------------------------------------------
```

9.2.5. The Output Graph

The next part is the graph that shows how memory consumption occurred as the program executed:

```

    KB
19.63^                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                       #
     |                                                                      :#
     |                                                                      :#
     |                                                                      :#
   0 +----------------------------------------------------------------------->ki     0


Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

Why is most of the graph empty, with only a couple of bars at the very end? By default, Massif uses "instructions executed" as the unit of time. For very short-run programs such as the example, most of the executed instructions involve the loading and dynamic linking of the program. The execution of `main` (and thus the heap allocations) only occur at the very end. For a short-running program like this, we can use the `--time-unit=B` option to specify that we want the time unit to instead be the number of bytes allocated/deallocated on the heap and stack(s).

If we re-run the program under Massif with this option, and then re-run ms_print, we get this more useful graph:

```

19.63^                                               ###
     |                                               #
     |                                               #  ::
     |                                               #  : :::
     |                                      :::::::::#  : :  ::
     |                                      :        #  : :  : ::
     |                                      :        #  : :  : : :::
     |                                      :        #  : :  : : :  ::
     |                            :::::::::::        #  : :  : : :  : :::
```

```
     |                            :         :        #  : :  : : :  : :  ::
     |                        :::::         :        #  : :  : : :  : :  : ::
     |                     @@@:   :         :        #  : :  : : :  : :  : : @
     |                   ::@  :   :         :        #  : :  : : :  : :  : : @
     |                :::: @  :   :         :        #  : :  : : :  : :  : : @
     |              :::  : @  :   :         :        #  : :  : : :  : :  : : @
     |            ::: :  : @  :   :         :        #  : :  : : :  : :  : : @
     |         :::: : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |       :::  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |    :::: :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
     |  :::  : :  : : :  : @  :   :         :        #  : :  : : :  : :  : : @
   0 +----------------------------------------------------------------------->KB     0

Number of snapshots: 25
 Detailed snapshots: [9, 14 (peak), 24]
```

The size of the graph can be changed with ms_print's `--x` and `--y` options. Each vertical bar represents a snapshot, i.e. a measurement of the memory usage at a certain point in time. If the next snapshot is more than one column away, a horizontal line of characters is drawn from the top of the snapshot to just before the next snapshot column. The text at the bottom show that 25 snapshots were taken for this program, which is one per heap allocation/deallocation, plus a couple of extras. Massif starts by taking snapshots for every heap allocation/deallocation, but as a program runs for longer, it takes snapshots less frequently. It also discards older snapshots as the program goes on; when it reaches the maximum number of snapshots (100 by default, although changeable with the `--max-snapshots` option) half of them are deleted. This means that a reasonable number of snapshots are always maintained.

Most snapshots are *normal*, and only basic information is recorded for them. Normal snapshots are represented in the graph by bars consisting of ':' characters.

Some snapshots are *detailed*. Information about where allocations happened are recorded for these snapshots, as we will see shortly. Detailed snapshots are represented in the graph by bars consisting of '@' characters. The text at the bottom show that 3 detailed snapshots were taken for this program (snapshots 9, 14 and 24). By default, every 10th snapshot is detailed, although this can be changed via the `--detailed-freq` option.

Finally, there is at most one *peak* snapshot. The peak snapshot is a detailed snapshot, and records the point where memory consumption was greatest. The peak snapshot is represented in the graph by a bar consisting of '#' characters. The text at the bottom shows that snapshot 14 was the peak.

Massif's determination of when the peak occurred can be wrong, for two reasons.

- Peak snapshots are only ever taken after a deallocation happens. This avoids lots of unnecessary peak snapshot recordings (imagine what happens if your program allocates a lot of heap blocks in succession, hitting a new peak every time). But it means that if your program never deallocates any blocks, no peak will be recorded. It also means that if your program does deallocate blocks but later allocates to a higher peak without subsequently deallocating, the reported peak will be too low.
- Even with this behaviour, recording the peak accurately is slow. So by default Massif records a peak whose size is within 1% of the size of the true peak. This inaccuracy in the peak measurement can be changed with the `--peak-inaccuracy` option.

The following graph is from an execution of Konqueror, the KDE web browser. It shows what graphs for larger programs look like.

```
    MB
3.952^                                                                    #
     |                                                                   @#:
```

```
     |                                                                 :@@#:
     |                                                            @@::::@@#:
     |                                                            @ :: :@@#::
     |                                                          @@@ :: :@@#::
     |                                                       @@:@@@ :: :@@#::
     |                                                    :::@ :@@@ :: :@@#::
     |                                                    : :@ :@@@ :: :@@#::
     |                                                  :@: :@ :@@@ :: :@@#::
     |                                                @@:@: :@ :@@@ :: :@@#:::
     |                           :       ::         ::@@:@: :@ :@@@ :: :@@#:::
     |                        :@@:    ::::: ::::@@@:::@@:@: :@ :@@@ :: :@@#:::
     |                     ::::@@:  ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                    @: ::@@:  ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                    @: ::@@:  ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                    @: ::@@:::::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |                ::@@@: ::@@:: ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |             :::::@ @: ::@@:: ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
     |           @@:::::@ @: ::@@:: ::: ::::::: @  :::@@:@: :@ :@@@ :: :@@#:::
   0 +---------------------------------------------------------------------->Mi
     0                                                                   626.4

Number of snapshots: 63
 Detailed snapshots: [3, 4, 10, 11, 15, 16, 29, 33, 34, 36, 39, 41,
                      42, 43, 44, 49, 50, 51, 53, 55, 56, 57 (peak)]
```

Note that the larger size units are KB, MB, GB, etc. As is typical for memory measurements, these are based on a multiplier of 1024, rather than the standard SI multiplier of 1000. Strictly speaking, they should be written KiB, MiB, GiB, etc.

9.2.6. The Snapshot Details

Returning to our example, the graph is followed by the detailed information for each snapshot. The first nine snapshots are normal, so only a small amount of information is recorded for each one:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1          1,008            1,008            1,000             8            0
  2          2,016            2,016            2,000            16            0
  3          3,024            3,024            3,000            24            0
  4          4,032            4,032            4,000            32            0
  5          5,040            5,040            5,000            40            0
  6          6,048            6,048            6,000            48            0
  7          7,056            7,056            7,000            56            0
  8          8,064            8,064            8,000            64            0
```

Each normal snapshot records several things.

- Its number.
- The time it was taken. In this case, the time unit is bytes, due to the use of `--time-unit=B`.
- The total memory consumption at that point.

- The number of useful heap bytes allocated at that point. This reflects the number of bytes asked for by the program.
- The number of extra heap bytes allocated at that point. This reflects the number of bytes allocated in excess of what the program asked for. There are two sources of extra heap bytes.

  First, every heap block has administrative bytes associated with it. The exact number of administrative bytes depends on the details of the allocator. By default Massif assumes 8 bytes per block, as can be seen from the example, but this number can be changed via the `--heap-admin` option.

  Second, allocators often round up the number of bytes asked for to a larger number, usually 8 or 16. This is required to ensure that elements within the block are suitably aligned. If N bytes are asked for, Massif rounds N up to the nearest multiple of the value specified by the `--alignment` option.
- The size of the stack(s). By default, stack profiling is off as it slows Massif down greatly. Therefore, the stack column is zero in the example. Stack profiling can be turned on with the `--stacks=yes` option.

The next snapshot is detailed. As well as the basic counts, it gives an allocation tree which indicates exactly which pieces of code were responsible for allocating heap memory:

```
  9          9,072            9,072            9,000            72            0
99.21% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.21% (9,000B) 0x804841A: main (example.c:20)
```

The allocation tree can be read from the top down. The first line indicates all heap allocation functions such as `malloc` and C++ `new`. All heap allocations go through these functions, and so all 9,000 useful bytes (which is 99.21% of all allocated bytes) go through them. But how were `malloc` and new called? At this point, every allocation so far has been due to line 20 inside `main`, hence the second line in the tree. The -> indicates that main (line 20) called `malloc`.

Let's see what the subsequent output shows happened next:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,080           10,080           10,000            80            0
 11         12,088           12,088           12,000            88            0
 12         16,096           16,096           16,000            96            0
 13         20,104           20,104           20,000           104            0
 14         20,104           20,104           20,000           104            0
99.48% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.74% (10,000B) 0x804841A: main (example.c:20)
|
->39.79% (8,000B) 0x80483C2: g (example.c:5)
| ->19.90% (4,000B) 0x80483E2: f (example.c:11)
| | ->19.90% (4,000B) 0x8048431: main (example.c:23)
| |
| ->19.90% (4,000B) 0x8048436: main (example.c:25)
|
->09.95% (2,000B) 0x80483DA: f (example.c:10)
  ->09.95% (2,000B) 0x8048431: main (example.c:23)
```

The first four snapshots are similar to the previous ones. But then the global allocation peak is reached, and a detailed snapshot (number 14) is taken. Its allocation tree shows that 20,000B of useful heap memory has been allocated, and the lines and arrows indicate that this is from three different code locations: line 20, which is responsible for 10,000B (49.74%); line 5, which is responsible for 8,000B (39.79%); and line 10, which is responsible for 2,000B (9.95%).

We can then drill down further in the allocation tree. For example, of the 8,000B asked for by line 5, half of it was due to a call from line 11, and half was due to a call from line 25.

In short, Massif collates the stack trace of every single allocation point in the program into a single tree, which gives a complete picture at a particular point in time of how and why all heap memory was allocated.

Note that the tree entries correspond not to functions, but to individual code locations. For example, if function `A` calls `malloc`, and function `B` calls `A` twice, once on line 10 and once on line 11, then the two calls will result in two distinct stack traces in the tree. In contrast, if `B` calls `A` repeatedly from line 15 (e.g. due to a loop), then each of those calls will be represented by the same stack trace in the tree.

Note also that each tree entry with children in the example satisfies an invariant: the entry's size is equal to the sum of its children's sizes. For example, the first entry has size 20,000B, and its children have sizes 10,000B, 8,000B, and 2,000B. In general, this invariant almost always holds. However, in rare circumstances stack traces can be malformed, in which case a stack trace can be a sub-trace of another stack trace. This means that some entries in the tree may not satisfy the invariant -- the entry's size will be greater than the sum of its children's sizes. This is not a big problem, but could make the results confusing. Massif can sometimes detect when this happens; if it does, it issues a warning:

```
Warning: Malformed stack trace detected.  In Massif's output,
         the size of an entry's child entries may not sum up
         to the entry's size as they normally do.
```

However, Massif does not detect and warn about every such occurrence. Fortunately, malformed stack traces are rare in practice.

Returning now to ms_print's output, the final part is similar:

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 15         21,112           19,096           19,000            96            0
 16         22,120           18,088           18,000            88            0
 17         23,128           17,080           17,000            80            0
 18         24,136           16,072           16,000            72            0
 19         25,144           15,064           15,000            64            0
 20         26,152           14,056           14,000            56            0
 21         27,160           13,048           13,000            48            0
 22         28,168           12,040           12,000            40            0
 23         29,176           11,032           11,000            32            0
 24         30,184           10,024           10,000            24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x80483C2: g (example.c:5)
| ->39.90% (4,000B) 0x80483E2: f (example.c:11)
| | ->39.90% (4,000B) 0x8048431: main (example.c:23)
| |
| ->39.90% (4,000B) 0x8048436: main (example.c:25)
|
->19.95% (2,000B) 0x80483DA: f (example.c:10)
| ->19.95% (2,000B) 0x8048431: main (example.c:23)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

The final detailed snapshot shows how the heap looked at termination. The 00.00% entry represents the code locations for which memory was allocated and then freed (line 20 in this case, the memory for which was freed on line 28).

However, no code location details are given for this entry; by default, Massif only records the details for code locations responsible for more than 1% of useful memory bytes, and ms_print likewise only prints the details for code locations responsible for more than 1%. The entries that do not meet this threshold are aggregated. This avoids filling up the output with large numbers of unimportant entries. The thresholds can be changed with the `--threshold` option that both Massif and ms_print support.

9.2.7. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent.

If the output file format string (controlled by `--massif-out-file`) does not contain `%p`, then the outputs from the parent and child will be intermingled in a single output file, which will almost certainly make it unreadable by ms_print.

9.2.8. Measuring All Memory in a Process

It is worth emphasising that by default Massif measures only heap memory, i.e. memory allocated with `malloc`, `calloc`, `realloc`, `memalign`, `new`, `new[]`, and a few other, similar functions. (And it can optionally measure stack memory, of course.) This means it does *not* directly measure memory allocated with lower-level system calls such as `mmap`, `mremap`, and `brk`.

Heap allocation functions such as `malloc` are built on top of these system calls. For example, when needed, an allocator will typically call `mmap` to allocate a large chunk of memory, and then hand over pieces of that memory chunk to the client program in response to calls to `malloc` et al. Massif directly measures only these higher-level `malloc` et al calls, not the lower-level system calls.

Furthermore, a client program may use these lower-level system calls directly to allocate memory. By default, Massif does not measure these. Nor does it measure the size of code, data and BSS segments. Therefore, the numbers reported by Massif may be significantly smaller than those reported by tools such as `top` that measure a program's total size in memory.

However, if you wish to measure *all* the memory used by your program, you can use the `--pages-as-heap=yes`. When this option is enabled, Massif's normal heap block profiling is replaced by lower-level page profiling. Every page allocated via `mmap` and similar system calls is treated as a distinct block. This means that code, data and BSS segments are all measured, as they are just memory pages. Even the stack is measured, since it is ultimately allocated (and extended when necessary) via `mmap`; for this reason `--stacks=yes` is not allowed in conjunction with `--pages-as-heap=yes`.

After `--pages-as-heap=yes` is used, ms_print's output is mostly unchanged. One difference is that the start of each detailed snapshot says:

```
(page allocation syscalls) mmap/mremap/brk, --alloc-fns, etc.
```

instead of the usual:

```
(heap allocation functions) malloc/new/new[], --alloc-fns, etc.
```

The stack traces in the output may be more difficult to read, and interpreting them may require some detailed understanding of the lower levels of a program like the memory allocators. But for some programs having the full information about memory usage can be very useful.

9.2.9. Acting on Massif's Information

Massif's information is generally fairly easy to act upon. The obvious place to start looking is the peak snapshot.

It can also be useful to look at the overall shape of the graph, to see if memory usage climbs and falls as you expect; spikes in the graph might be worth investigating.

The detailed snapshots can get quite large. It is worth viewing them in a very wide window. It's also a good idea to view them with a text editor. That makes it easy to scroll up and down while keeping the cursor in a particular column, which makes following the allocation chains easier.

9.3. Using massif-visualizer

massif-visualizer is a graphical viewer for Massif data that is often easier to use than ms_print. massif-visualizer is not shipped within Valgrind, but is available in various places online.

9.4. Massif Command-line Options

Massif-specific command-line options are:

--heap=<yes|no> [default: yes]

Specifies whether heap profiling should be done.

--heap-admin=<size> [default: 8]

If heap profiling is enabled, gives the number of administrative bytes per block to use. This should be an estimate of the average, since it may vary. For example, the allocator used by glibc on Linux requires somewhere between 4 to 15 bytes per block, depending on various factors. That allocator also requires admin space for freed blocks, but Massif cannot account for this.

--stacks=<yes|no> [default: no]

Specifies whether stack profiling should be done. This option slows Massif down greatly, and so is off by default. Note that Massif assumes that the main stack has size zero at start-up. This is not true, but doing otherwise accurately is difficult. Furthermore, starting at zero better indicates the size of the part of the main stack that a user program actually has control over.

If you give at least 4 -v verbosity arguments, then massif produces a trace for each stack increase and decrease. The stack increase trace contains the IP address that increased the stack. Note that to get fully precise IP address, you must specify the options -px-default=unwindregs-at-mem-access --px-file-backed=unwindregs-at-mem-access.

--pages-as-heap=<yes|no> [default: no]

Tells Massif to profile memory at the page level rather than at the malloc'd block level. See above for details.

--depth=<number> [default: 30]

Maximum depth of the allocation trees recorded for detailed snapshots. Increasing it will make Massif run somewhat more slowly, use more memory, and produce bigger output files.

--alloc-fn=<name>

Functions specified with this option will be treated as though they were a heap allocation function such as malloc. This is useful for functions that are wrappers to malloc or new, which can fill up the allocation trees with uninteresting information. This option can be specified multiple times on the command line, to name multiple functions.

Note that the named function will only be treated this way if it is the top entry in a stack trace, or just below another function treated this way. For example, if you have a function `malloc1` that wraps `malloc`, and `malloc2` that wraps `malloc1`, just specifying `--alloc-fn=malloc2` will have no effect. You need to specify `--alloc-fn=malloc1` as well. This is a little inconvenient, but the reason is that checking for allocation functions is slow, and it saves a lot of time if Massif can stop looking through the stack trace entries as soon as it finds one that doesn't match rather than having to continue through all the entries.

Note that C++ names are demangled. Note also that overloaded C++ names must be written in full. Single quotes may be necessary to prevent the shell from breaking them up. For example:

```
--alloc-fn='operator new(unsigned, std::nothrow_t const&)'
```

Arguments of type `size_t` need to be replaced with `unsigned long` on 64bit platforms and `unsigned` on 32bit platforms.

`--alloc-fn` will work with inline functions. Inline function names are not mangled, which means that you only need to provide the function name and not the argument list.

`--alloc-fn` does not support wildcards.

`--ignore-fn=<name>`

Any direct heap allocation (i.e. a call to `malloc`, `new`, etc, or a call to a function named by an `--alloc-fn` option) that occurs in a function specified by this option will be ignored. This is mostly useful for testing purposes. This option can be specified multiple times on the command line, to name multiple functions.

Any `realloc` of an ignored block will also be ignored, even if the `realloc` call does not occur in an ignored function. This avoids the possibility of negative heap sizes if ignored blocks are shrunk with `realloc`.

The rules for writing C++ function names are the same as for `--alloc-fn` above.

`--threshold=<m.n> [default: 1.0]`

The significance threshold for heap allocations, as a percentage of total memory size. Allocation tree entries that account for less than this will be aggregated. Note that this should be specified in tandem with ms_print's option of the same name.

`--peak-inaccuracy=<m.n> [default: 1.0]`

Massif does not necessarily record the actual global memory allocation peak; by default it records a peak only when the global memory allocation size exceeds the previous peak by at least 1.0%. This is because there can be many local allocation peaks along the way, and doing a detailed snapshot for every one would be expensive and wasteful, as all but one of them will be later discarded. This inaccuracy can be changed (even to 0.0%) via this option, but Massif will run drastically slower as the number approaches zero.

`--time-unit=<i|ms|B> [default: i]`

The time unit used for the profiling. There are three possibilities: instructions executed (i), which is good for most cases; real (wallclock) time (ms, i.e. milliseconds), which is sometimes useful; and bytes allocated/deallocated on the heap and/or stack (B), which is useful for very short-run programs, and for testing purposes, because it is the most reproducible across different machines.

`--detailed-freq=<n> [default: 10]`

Frequency of detailed snapshots. With `--detailed-freq=1`, every snapshot is detailed.

`--max-snapshots=<n> [default: 100]`

The maximum number of snapshots recorded. If set to N, for all programs except very short-running ones, the final number of snapshots will be between N/2 and N.

`--massif-out-file=<file> [default: massif.out.%p]`

Write the profile data to `file` rather than to the default output file, `massif.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

9.5. Massif Monitor Commands

The Massif tool provides monitor commands handled by the Valgrind gdbserver (see Monitor command handling by the Valgrind gdbserver). Valgrind python code provides GDB front end commands giving an easier usage of the massif monitor commands (see GDB front end commands for Valgrind gdbserver monitor commands). To launch a massif monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `massif` command (or the shorter aliases `ms`). Using the massif GDB front end command provide a more flexible usage, such as auto-completion of the command by GDB. In GDB, you can use `help massif` to get help about the massif front end monitor commands and you can use `apropos massif` to get all the commands mentionning the word "massif" in their name or on-line help.

- `snapshot [<filename>]` requests to take a snapshot and save it in the given <filename> (default massif.vgdb.out).
- `detailed_snapshot [<filename>]` requests to take a detailed snapshot and save it in the given <filename> (default massif.vgdb.out).
- `all_snapshots [<filename>]` requests to take all captured snapshots so far and save them in the given <filename> (default massif.vgdb.out).
- `xtmemory [<filename> default xtmemory.kcg.%p.%n]` requests Massif tool to produce an xtree heap memory report. See Execution Trees for a detailed explanation about execution trees.

9.6. Massif Client Requests

Massif does not have a `massif.h` file, but it does implement two of the core client requests: `VALGRIND_MALLOCLIKE_BLOCK` and `VALGRIND_FREELIKE_BLOCK`; they are described in The Client Request mechanism.

9.7. ms_print Command-line Options

ms_print's options are:

`-h --help`

Show the help message.

`--version`

Show the version number.

`--threshold=<m.n> [default: 1.0]`

Same as Massif's `--threshold` option, but applied after profiling rather than during.

`--x=<4..1000> [default: 72]`

Width of the graph, in columns.

`--y=<4..1000> [default: 20]`

Height of the graph, in rows.

9.8. Massif's Output File Format

Massif's file format is plain text (i.e. not binary) and deliberately easy to read for both humans and machines. Nonetheless, the exact format is not described here. This is because the format is currently very Massif-specific. In the future we hope to make the format more general, and thus suitable for possible use with other tools. Once this has been done, the format will be documented here.

10. DHAT: a dynamic heap analysis tool

To use this tool, you must specify `--tool=dhat` on the Valgrind command line.

10.1. Overview

DHAT is primarily a tool for examining how programs use their heap allocations.

It tracks the allocated blocks, and inspects every memory access to find which block, if any, it is to. It presents, on a program point basis, information about these blocks such as sizes, lifetimes, numbers of reads and writes, and read and write patterns.

Using this information it is possible to identify program points with the following characteristics:

- potential process-lifetime leaks: blocks allocated by the point just accumulate, and are freed only at the end of the run.
- excessive turnover: points which chew through a lot of heap, even if it is not held onto for very long
- excessively transient: points which allocate very short lived blocks
- useless or underused allocations: blocks which are allocated but not completely filled in, or are filled in but not subsequently read.
- blocks with inefficient layout -- areas never accessed, or with hot fields scattered throughout the block.

As with the Massif heap profiler, DHAT measures program progress by counting instructions, and so presents all age/time related figures as instruction counts. This sounds a little odd at first, but it makes runs repeatable in a way which is not possible if CPU time is used.

DHAT also has support for copy profiling and ad hoc profiling. These are described below.

10.2. Using DHAT

First off, as for normal Valgrind use, you probably want to compile with debugging info (the `-g` option). But by contrast with normal Valgrind use, you probably do want to turn optimisation on, since you should profile your program as it will be normally run.

Second, you need to run your program under DHAT to gather the profiling information. You might need to reduce the `--num-callers` value to get reasonably-sized output files, especially if you are profiling a large program; some trial and error might be needed to find a good value.

Finally, you need to use DHAT's viewer (in a web browser) to get a detailed presentation of that information.

10.2.1. Running DHAT

To run DHAT on a program `prog`, run:

```
valgrind --tool=dhat prog
```

The program will execute (slowly). Upon completion, summary statistics that look like this will be printed:

```
==11514== Total:     823,849,731 bytes in 3,929,133 blocks
==11514== At t-gmax: 133,485,082 bytes in 436,521 blocks
==11514== At t-end:  258,002 bytes in 2,129 blocks
==11514== Reads:     2,807,182,810 bytes
==11514== Writes:    1,149,617,086 bytes
```

The first line shows how many heap blocks and bytes were allocated over the entire execution.

The second line shows how many heap blocks and bytes were alive at `t-gmax`, i.e. the time when the heap size reached its global maximum (as measured in bytes).

The third line shows how many heap blocks and bytes were alive at `t-end`, i.e. the end of execution. In other words, how many blocks and bytes were not explicitly freed.

The fourth and fifth lines show how many bytes within heap blocks were read and written during the entire execution.

These lines are moderately interesting at best. More useful information can be seen with DHAT's viewer.

10.2.2. Output File

As well as printing summary information, DHAT also writes more detailed profiling information to a file. By default this file is named `dhat.out.<pid>` (where `<pid>` is the program's process ID), but its name can be changed with the `--dhat-out-file` option. This file is JSON, and intended to be viewed by DHAT's viewer, which is described in the next section.

The default `.<pid>` suffix on the output file name serves two purposes. Firstly, it means you don't have to rename old log files that you don't want to overwrite. Secondly, and more importantly, it allows correct profiling with the `--trace-children=yes` option of programs that spawn child processes.

The output file can be big, many megabytes for large applications built with full debugging information.

10.3. DHAT's Viewer

DHAT's viewer can be run in a web browser by loading the file `dh_view.html`. Use the "Load" button to choose a DHAT output file to view.

If loading takes a long time, it might be worth re-running DHAT with a smaller `--num-callers` value to reduce the stack depths, because this can significantly reduce the size of DHAT's output files.

10.3.1. The Output Header

The first part of the output shows the mode, program command and process ID. For example:

```
Invocation {
  Mode:    heap
  Command: /home/njn/moz/rust0/build/x86_64-unknown-linux-gnu/stage2/bin/rustc --crate-nam
  PID:     18816
}
```

The second part of the output shows the `t-gmax` and `t-end` values again. For example:

```
Times {
  t-gmax: 8,138,210,673 instrs (86.92% of program duration)
  t-end:  9,362,544,994 instrs
}
```

10.3.2. The PP Tree

The third part of the output is the largest and most interesting part, showing the program point (PP) tree.

10.3.2.1. Structure

The following image shows a screenshot of part of a PP tree. The font is very small because this screenshot is intended to demonstrate the high-level structure of the tree rather than the details within the text. (It is also slightly out-of-date, and doesn't quite match the current output produced by DHAT's viewer.)

Like any tree, it has a root node, leaf nodes, and non-leaf nodes. The structure of the tree is shown by the lines connecting nodes. Child nodes are beneath their parent and indented one level.

The sub-trees beneath a non-leaf node can be collapsed or expanded by clicking on the node. It is useful to collapse sub-trees that you aren't interested in.

Colours are meaningful, and are intended to ease tree navigation, but the information they represent is also present within the text. (This means that colour-blind users are not denied any information.)

Each leaf node is coloured green. Each non-leaf node is coloured blue and has a down arrow (#) next to it when its sub-tree is expanded. Each non-leaf node is coloured yellow and has a left arrow (#) next to it when its sub-tree is collapsed.

The shade of green, blue or yellow used for a node indicate its significance. Darker shades represent greater significance (in terms of bytes or blocks).

Note that the entire output is text, even the arrows and lines connecting nodes. This means you can copy and paste any part of the output easily into an email, bug report, etc.

10.3.2.2. The Root Node

The root node looks like this:

```
PP 1/1 (25 children) {
  Total:     1,355,253,987 bytes (100%, 67,454.81/Minstr) in 5,943,417 blocks (100%, 295.8
  At t-gmax: 423,930,307 bytes (100%) in 1,575,682 blocks (100%), avg size 269.05 bytes
  At t-end:  258,002 bytes (100%) in 2,129 blocks (100%), avg size 121.18 bytes
  Reads:     5,478,606,988 bytes (100%, 272,685.7/Minstr), 4.04/byte
  Writes:    2,040,294,800 bytes (100%, 101,551.22/Minstr), 1.51/byte
  Allocated at {
    #0: [root]
  }
}
```

The root node covers the entire execution. The information is a superset of the information shown when DHAT ran, adding details such as allocation rates, average block sizes, block lifetimes, and read and write ratios. The next example will explain these in more detail.

10.3.2.3. Interior Nodes

PP nodes further down the tree show information about a subset of allocations. For example:

```
PP 1.1/25 (2 children) {
  Total:     54,533,440 bytes (4.02%, 2,714.28/Minstr) in 458,839 blocks (7.72%, 22.84/Min
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     15,993,012 bytes (0.29%, 796.02/Minstr), 0.29/byte
  Writes:    20,974,752 bytes (1.03%, 1,043.97/Minstr), 0.38/byte
  Allocated at {
    #1: 0x95CACC9: alloc (alloc.rs:72)
    #2: 0x95CACC9: alloc (alloc.rs:148)
    #3: 0x95CACC9: reserve_internal<syntax::tokenstream::TokenStream,alloc::alloc::Global>
    #4: 0x95CACC9: reserve<syntax::tokenstream::TokenStream,alloc::alloc::Global> (raw_vec
    #5: 0x95CACC9: reserve<syntax::tokenstream::TokenStream> (vec.rs:460)
    #6: 0x95CACC9: push<syntax::tokenstream::TokenStream> (vec.rs:989)
    #7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
    #8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::StringRea
  }
}
```

The first line indicates the node's position in the tree. The `1.1` is a unique identifier for the node and also says that it is the first child node `1` (which is the root). The `/25` says that it is one of 25 children, i.e. it has 24 siblings. The `(2 children)` says that this node node has two children of its own.

Allocations are aggregated by their allocation stack trace. The `Allocated at` section shows the allocation stack trace that is shared by all the blocks covered by this node.

The `Total` line shows that this node accounts for 4.02% of all bytes allocated during execution, and 7.72% of all blocks. These percentages are useful for comparing the significance of different nodes within a single profile; a PP that accounts for 10% of bytes allocated is likely to be more interesting than one that accounts for 2%.

The `Total` line also shows allocation rates, measured in bytes and blocks per million instructions. These rates are useful for comparing the significance of nodes across profiles made with different workloads.

Finally, the `Total` line shows the average size and lifetimes of these blocks.

The `At t-gmax` line says shows that no blocks from this PP were alive when the global heap peak occurred. In other words, these blocks do not contribute at all to the global heap peak.

The `At t-end` line shows that no blocks were from this PP were alive at shutdown. In other words, all those blocks were explicitly freed before termination.

The `Reads` and `Writes` lines show how many bytes were read within this PP's blocks, the fraction this represents of all heap reads, and the read rate. Finally, it shows the read ratio, which is the number of reads per byte. In this case the number is 0.29, which is quite low -- if no byte was read twice, then only 29% of the allocated bytes, which means that at least 71% of the bytes were never read! This suggests that the blocks are being underutilized and might be worth optimizing.

The `Writes` lines is similar to the `Reads` line. In this case, at most 38% of the bytes are ever written, and at least 62% of the bytes were never written.

The `Reads` and `Writes` measurements suggest that the blocks are being under-utilised and might be worth optimizing. Having said that, this kind of under-utilisation is common in data structures that grow, such as vectors and hash tables, and isn't always fixable.

10.3.2.4. Leaf Nodes

This is a leaf node:

```
PP 1.1.1.1/2 {
  Total:     31,460,928 bytes (2.32%, 1,565.9/Minstr) in 262,171 blocks (4.41%, 13.05/Mins
  Max:       16,779,136 bytes in 65,543 blocks, avg size 256 bytes
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     5,964,704 bytes (0.11%, 296.88/Minstr), 0.19/byte
  Writes:    10,487,200 bytes (0.51%, 521.98/Minstr), 0.33/byte
  Allocated at {
    ^1: 0x95CACC9: alloc (alloc.rs:72)
    ^2: 0x95CACC9: alloc (alloc.rs:148)
    ^3: 0x95CACC9: reserve_internal<syntax::tokenstream::TokenStream,alloc::alloc::Global>
    ^4: 0x95CACC9: reserve<syntax::tokenstream::TokenStream,alloc::alloc::Global> (raw_vec
    ^5: 0x95CACC9: reserve<syntax::tokenstream::TokenStream> (vec.rs:460)
    ^6: 0x95CACC9: push<syntax::tokenstream::TokenStream> (vec.rs:989)
    ^7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
    ^8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::StringRea
    ^9: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
    ^10: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::StringRe
    #11: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
    #12: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::StringRe
  }
}
```

The `1.1.1.1/2` indicates that this node is a great-grandchild of the root; is the first grandchild of the node in the previous example; and has no children.

Leaf nodes contain an additional `Max` line, indicating the peak memory use for the blocks covered by this PP. (This peak may have occurred at a time other than `t-gmax`.) In this case, 31,460,298 bytes were allocated from this PP, but the maximum size alive at once was 16,779,136 bytes.

Stack frames that begin with a `^` rather than a `#` are copied from ancestor nodes. (In this example, the first 8 frames are identical to those from the node in the previous example.) These frames could be found by tracing back through ancestor nodes, but that can be annoying, which is why they are duplicated. This also means that each node makes complete sense on its own.

10.3.2.5. Access Counts

If all blocks covered by a PP node have the same size, an additional `Accesses` field will be present. It indicates how the reads and writes within these blocks were distributed. For example:

```
Total:     8,388,672 bytes (0.62%, 417.53/Minstr) in 262,146 blocks (4.41%, 13.05/Minstr),
At t-gmax: 8,388,672 bytes (1.98%) in 262,146 blocks (16.64%), avg size 32 bytes
At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
Reads:     9,109,682 bytes (0.17%, 453.41/Minstr), 1.09/byte
Writes:    7,340,088 bytes (0.36%, 365.34/Minstr), 0.88/byte
Accesses: {
  [  0]  65547 7 8 4 65529 # # # 16 # # # 12 # # # # # # # # # # # 65542 # # # - - - -
}
```

Every block covered by this PP was 32 bytes. Within all of those blocks, byte 0 was accessed (read or written) 65,547 times, byte 1 was accessed 7 times, byte 2 was accessed 8 times, and so on.

The ditto symbol (#) means "same access count as the previous byte".

A dash (-) means "zero". (It is used instead of `0` because it makes unaccessed regions more easily identifiable.)

The infinity symbol (#, not present in this example) means "exceeded the maximum tracked count".

Block layout can often be inferred from counts. For example, these blocks probably have four separate byte-sized fields, followed by a four-byte field, and so on.

The size of the blocks that measure and display access counts is limited to 1024 bytes. This is done to limit the performance overhead and also to keep the size of the generated output reasonable. However, it is possible to override this limit using client requests. The use-case for this is to first run DHAT normally, and then identify any large blocks that you would like to further investigate with access count histograms. The client request is declared in `dhat/dhat.h` and is called `DHAT_HISTOGRAM_MEMORY`. The macro should be placed immediately after the call to the allocator, and use the pointer returned by the allocator.

```
// LargeStruct bigger than 1024 bytes
struct LargeStruct* ls = malloc(sizeof(struct LargeStruct));
DHAT_HISTOGRAM_MEMORY(ls);
```

The memory that can be profiled in this way with user requests has a further upper limit of 25kbytes. Be aware that the access counts will all be set to zero. This means that the access counts will not include any reads or writes performed during initialisation. An example where this will happen are uses of C++ `new` with user-defined constructors.

Access counts can be useful for identifying data alignment holes or other layout inefficiencies.

10.3.2.6. Aggregate Nodes

The PP tree is very large and many nodes represent tiny numbers of blocks and bytes. Therefore, DHAT's viewer aggregates insignificant nodes like this:

```
PP 1.14.2/2 {
  Total:     5,175 blocks (0.09%, 0.26/Minstr)
  Allocated at {
    [5 insignificant]
  }
}
```

Much of the detail is stripped away, leaving only basic measurements, along with an indication of how many nodes were aggregated together (5 in this case).

10.3.3. The Output Footer

Below the PP tree is a line like this:

```
PP significance threshold: total >= 59,434.17 blocks (1%)
```

It shows the function used to determine if a PP node is significant. All nodes that don't satisfy this function are aggregated. It is occasionally useful if you don't understand why a PP node has been aggregated. The exact threshold depends on the sort metric (see below).

Finally, the bottom of the page shows a legend that explains some of the terms, abbreviations and symbols used in the output.

10.3.4. Sort Metrics

The order in which sub-trees are sorted can be changed via the "Sort metric" drop-down menu at the top of DHAT's viewer. Different sort metrics can be useful for finding different things. Some sort metrics also incorporate some filtering, so that only nodes meeting a particular criteria are shown.

Total (bytes)

> The total number of bytes allocated during the execution. Highly useful for evaluating heap churn, though not quite as useful as "Total (blocks)".

Total (blocks)

> The total number of blocks allocated during the execution. Highly useful for evaluating heap churn; reducing the number of calls to the allocator can significantly speed up a program. This is the default sort metric.

Total (blocks), tiny

> Like "Total (blocks)", but shows only very small blocks. Moderately useful, because such blocks are often easy to avoid allocating.

Total (blocks), short-lived

> Like "Total (blocks)", but shows only very short-lived blocks. Moderately useful, because such blocks are often easy to avoid allocating.

Total (bytes), zero reads or zero writes

> Like "Total (bytes)", but shows only blocks that are never read or never written to (or both). Highly useful, because such blocks indicate poor use of memory and are often easy to avoid allocating. For example, sometimes a block is allocated and written to but then only read if a condition C is true; in that case, it may be possible to delay creating the block until condition C is true. Alternatively, sometimes blocks are created and never used; such blocks are trivial to remove.

Total (blocks), zero reads or zero writes

> Like "Total (bytes), zero reads or zero writes" but for blocks. Highly useful.

Total (bytes), low-access

> Like "Total (bytes)", but shows only blocks that have low numbers of reads or low numbers of writes (or both). Moderately useful, because such blocks indicate poor use of memory.

Total (blocks), low-access

> Like "Total (bytes), low-access", but for blocks.

At t-gmax (bytes)

> This shows the breakdown of memory at the point of peak heap memory usage. Highly useful for reducing peak memory usage.

At t-end (bytes)

> This shows the breakdown of memory at program termination. Highly useful for identifying process-lifetime leaks.

Reads (bytes)

The number of bytes read within heap blocks. Occasionally useful.

Reads (bytes), high-access

Like "Reads (bytes)", but only shows blocks with high read ratios. Occasionally useful for identifying hot areas of memory.

Writes (bytes)

Like "Reads (bytes)", but for writes. Occasionally useful.

Writes (bytes), high-access

Like "Reads (bytes), high-access", but for writes. Occasionally useful.

The values within a node that represent the chosen sort metric are shown in bold, so they stand out.

Here is part of a PP node found with "Total (blocks), tiny", showing blocks with an average size of only 8.67 bytes:

```
Total:     3,407,848 bytes (0.25%, 169.62/Minstr) in 393,214 blocks (6.62%, 19.57/Minstr),
```

Here is part of a PP node found with "Total (blocks), short-lived", showing blocks with an average lifetime of only 181.75 instructions:

```
Total:     23,068,584 bytes (1.7%, 1,148.19/Minstr) in 262,143 blocks (4.41%, 13.05/Minstr
```

Here is an example of a PP identified with "Total (blocks), zero reads or zero writes", showing blocks that are allocated but never touched:

```
Total:     7,339,920 bytes (0.54%, 365.33/Minstr) in 262,140 blocks (4.41%, 13.05/Minstr),
Max:       3,669,960 bytes in 131,070 blocks, avg size 28 bytes
At t-gmax: 3,336,400 bytes (0.79%) in 119,157 blocks (7.56%), avg size 28 bytes
At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
Reads:     0 bytes (0%, 0/Minstr), 0/byte
Writes:    0 bytes (0%, 0/Minstr), 0/byte
```

All the blocks identified by these PPs are good candidates for optimization.

10.4. Treatment of realloc

`realloc` is a tricky function and there are several different ways that DHAT could handle it.

Imagine a `malloc(100)` call followed by a `realloc(200)` call. This combination is considered to add two to the total block count, and 300 bytes to the total bytes count. (An alternative would be to only add one to the total block count, and 200 bytes to the total bytes count, as if a single `malloc(200)` call had occurred. While this would be defensible from a semantic point of view, it is silly from an operational point of view, because making two calls to allocator functions is more expensive than one call, and DHAT is a profiler that aims to help with runtime costs.)

Furthermore, the implicit copying of the 100 bytes is added to the reads and writes counts. Without this, the read and write counts would be under-measured and misleading.

However, DHAT only increases the current heap size by 100 bytes for this combination, and does not change the current block count. (As opposed to increasing the current heap size by 200 bytes and then decreasing it by 100 bytes.) As a result, it can only increase the global heap peak (if indeed, this results in a new peak) by 100 bytes.

Finally, the program point assigned to the block allocated by the `malloc(100)` call is retained once the block is reallocated. Which means that all 300 bytes are attributed to that program point, and no separate program point is created for the `realloc(200)` call. This may be surprising, but it has one large benefit.

Imagine some code that starts with an empty buffer, and then gradually adds data to that buffer from numerous different points in the code, reallocating the buffer each time it gets full. (E.g. code generation in a compiler might work this way.) With the described approach, the first heap block and all subsequent heap blocks are attributed to the same program point. While this is something of a lie -- the first program point isn't actually responsible for the other allocations -- it is arguably better than having the program points spread around in a distribution that unpredictably depends on whenever the reallocations were triggered.

10.5. Copy profiling

If DHAT is invoked with `--mode=copy`, instead of profiling heap operations (allocations and deallocations), it profiles copy operations, such as `memcpy`, `memmove`, `strcpy`, and `bcopy`. This is sometimes useful.

Here is an example PP node from this mode:

```
PP 1.1.2/5 (4 children) {
  Total:     1,210,925 bytes (10.03%, 4,358.66/Minstr) in 112,717 blocks (35.2%, 405.72/Mi
  Copied at {
    ^1: 0x4842524: memmove (vg_replace_strmem.c:1289)
    #2: 0x1F0A0D: copy_nonoverlapping<u8> (intrinsics.rs:1858)
    #3: 0x1F0A0D: copy_from_slice<u8> (mod.rs:2524)
    #4: 0x1F0A0D: spec_extend<u8> (vec.rs:2227)
    #5: 0x1F0A0D: extend_from_slice<u8> (vec.rs:1619)
    #6: 0x1F0A0D: push_str (string.rs:821)
    #7: 0x1F0A0D: write_str (string.rs:2418)
    #8: 0x1F0A0D: <&mut W as core::fmt::Write>::write_str (mod.rs:195)
  }
}
```

It is very similar to the PP nodes for heap profiling, but with less information, because copy profiling doesn't involve any tracking of memory regions with lifetimes.

10.6. Ad hoc profiling

If DHAT is invoked with `--mode=ad-hoc`, instead of profiling heap operations (allocations and deallocations), it profiles calls to the `DHAT_AD_HOC_EVENT` client request, which is declared in `dhat/dhat.h`.

Here is an example PP node from this mode:

```
PP 1.1.1.1/2 {
  Total:     30 units (17.65%, 115.97/Minstr) in 1 events (14.29%, 3.87/Minstr), avg size
  Occurred at {
    ^1: 0x109407: g (ad-hoc.c:4)
    ^2: 0x109425: f (ad-hoc.c:8)
    #3: 0x109497: main (ad-hoc.c:14)
```

```
  }
}
```

This kind of profiling is useful when you know a code path is hot but you want to know more about it.

For example, you might want to know which callsites of a hot function account for most of the calls. You could put a `DHAT_AD_HOC_EVENT(1);` call at the start of that function.

Alternatively, you might want to know the typical length of a vector in a hot location. You could put a `DHAT_AD_HOC_EVENT(len);` call at the appropriate location, when `len` is the length of the vector.

10.7. DHAT Command-line Options

DHAT-specific command-line options are:

`--dhat-out-file=<file>`

> Write the profile data to `file` rather than to the default output file, `dhat.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

`--mode=<heap|copy|ad-hoc> [default: heap]`

> The profiling mode: heap profiling, copy profiling, or ad hoc profiling.

Note that stacks by default have 12 frames. This may be more than necessary, in which case the `--num-callers` flag can be used to reduce the number, which may make DHAT run slightly faster.

11. Lackey: an example tool

To use this tool, you must specify --tool=lackey on the Valgrind command line.

11.1. Overview

Lackey is a simple Valgrind tool that does various kinds of basic program measurement. It adds quite a lot of simple instrumentation to the program's code. It is primarily intended to be of use as an example tool, and consequently emphasises clarity of implementation over performance.

11.2. Lackey Command-line Options

Lackey-specific command-line options are:

--basic-counts=<no|yes> [default: yes]

When enabled, Lackey prints the following statistics and information about the execution of the client program:

1. The number of calls to the function specified by the --fnname option (the default is main). If the program has had its symbols stripped, the count will always be zero.
2. The number of conditional branches encountered and the number and proportion of those taken.
3. The number of superblocks entered and completed by the program. Note that due to optimisations done by the JIT, this is not at all an accurate value.
4. The number of guest (x86, amd64, ppc, etc.) instructions and IR statements executed. IR is Valgrind's RISC-like intermediate representation via which all instrumentation is done.
5. Ratios between some of these counts.
6. The exit code of the client program.

--detailed-counts=<no|yes> [default: no]

When enabled, Lackey prints a table containing counts of loads, stores and ALU operations, differentiated by their IR types. The IR types are identified by their IR name ("I1", "I8", ... "I128", "F32", "F64", and "V128").

--trace-mem=<no|yes> [default: no]

When enabled, Lackey prints the size and address of almost every memory access made by the program. See the comments at the top of the file lackey/lk_main.c for details about the output format, how it works, and inaccuracies in the address trace. Note that this option produces immense amounts of output.

--trace-superblocks=<no|yes> [default: no]

When enabled, Lackey prints out the address of every superblock (a single entry, multiple exit, linear chunk of code) executed by the program. This is primarily of interest to Valgrind developers. See the comments at the top of the file lackey/lk_main.c for details about the output format. Note that this option produces large amounts of output.

--fnname=<name> [default: main]

Changes the function for which calls are counted when --basic-counts=yes is specified.

12. Nulgrind: the minimal Valgrind tool

To use this tool, you must specify `--tool=none` on the Valgrind command line.

12.1. Overview

Nulgrind is the simplest possible Valgrind tool. It performs no instrumentation or analysis of a program, just runs it normally. It is mainly of use for Valgrind's developers for debugging and regression testing.

Nonetheless you can run programs with Nulgrind. They will run roughly 5 times more slowly than normal, for no useful effect. Note that you need to use the option `--tool=none` to run Nulgrind (ie. not `--tool=nulgrind`).

13. BBV: an experimental basic block vector generation tool

To use this tool, you must specify `--tool=exp-bbv` on the Valgrind command line.

13.1. Overview

A basic block is a linear section of code with one entry point and one exit point. A *basic block vector* (BBV) is a list of all basic blocks entered during program execution, and a count of how many times each basic block was run.

BBV is a tool that generates basic block vectors for use with the SimPoint analysis tool. The SimPoint methodology enables speeding up architectural simulations by only running a small portion of a program and then extrapolating total behavior from this small portion. Most programs exhibit phase-based behavior, which means that at various times during execution a program will encounter intervals of time where the code behaves similarly to a previous interval. If you can detect these intervals and group them together, an approximation of the total program behavior can be obtained by only simulating a bare minimum number of intervals, and then scaling the results.

In computer architecture research, running a benchmark on a cycle-accurate simulator can cause slowdowns on the order of 1000 times, making it take days, weeks, or even longer to run full benchmarks. By utilizing SimPoint this can be reduced significantly, usually by 90-95%, while still retaining reasonable accuracy.

A more complete introduction to how SimPoint works can be found in the paper "Automatically Characterizing Large Scale Program Behavior" by T. Sherwood, E. Perelman, G. Hamerly, and B. Calder.

13.2. Using Basic Block Vectors to create SimPoints

To quickly create a basic block vector file, you will call Valgrind like this:

```
valgrind --tool=exp-bbv /bin/ls
```

In this case we are running on `/bin/ls`, but this can be any program. By default a file called `bb.out.PID` will be created, where PID is replaced by the process ID of the running process. This file contains the basic block vector. For long-running programs this file can be quite large, so it might be wise to compress it with gzip or some other compression program.

To create actual SimPoint results, you will need the SimPoint utility, available from the SimPoint webpage. Assuming you have downloaded SimPoint 3.2 and compiled it, create SimPoint results with a command like the following:

```
./SimPoint.3.2/bin/simpoint -inputVectorsGzipped \
    -loadFVFile bb.out.1234.gz \
    -k 5 -saveSimpoints results.simpts \
    -saveSimpointWeights results.weights
```

where bb.out.1234.gz is your compressed basic block vector file generated by BBV.

The SimPoint utility does random linear projection using 15-dimensions, then does k-mean clustering to calculate which intervals are of interest. In this example we specify 5 intervals with the -k 5 option.

The outputs from the SimPoint run are the `results.simpts` and `results.weights` files. The first holds the 5 most relevant intervals of the program. The seconds holds the weight to scale each interval by when extrapolating full-program behavior. The intervals and the weights can be used in conjunction with a simulator that supports fast-forwarding; you fast-forward to the interval of interest, collect stats for the desired interval length, then use statistics gathered in conjunction with the weights to calculate your results.

13.3. BBV Command-line Options

BBV-specific command-line options are:

`--bb-out-file=<name> [default: bb.out.%p]`

This option selects the name of the basic block vector file. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

`--pc-out-file=<name> [default: pc.out.%p]`

This option selects the name of the PC file. This file holds program counter addresses and function name info for the various basic blocks. This can be used in conjunction with the basic block vector file to fast-forward via function names instead of just instruction counts. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

`--interval-size=<number> [default: 100000000]`

This option selects the size of the interval to use. The default is 100 million instructions, which is a commonly used value. Other sizes can be used; smaller intervals can help programs with finer-grained phases. However smaller interval size can lead to accuracy issues due to warm-up effects (When fast-forwarding the various architectural features will be un-initialized, and it will take some number of instructions before they "warm up" to the state a full simulation would be at without the fast-forwarding. Large interval sizes tend to mitigate this.)

`--instr-count-only [default: no]`

This option tells the tool to only display instruction count totals, and to not generate the actual basic block vector file. This is useful for debugging, and for gathering instruction count info without generating the large basic block vector files.

13.4. Basic Block Vector File Format

The Basic Block Vector is dumped at fixed intervals. This is commonly done every 100 million instructions; the `--interval-size` option can be used to change this.

The output file looks like this:

```
T:45:1024 :189:99343
T:11:78573 :15:1353  :56:1
T:18:45 :12:135353 :56:78 314:4324263
```

Each new interval starts with a T. This is followed on the same line by a series of basic block and frequency pairs, one for each basic block that was entered during the interval. The format for each block/frequency pair is a colon, followed by a number that uniquely identifies the basic block, another colon, and then the frequency (which is the number of times the block was entered, multiplied by the number of instructions in the block). The pairs are separated from each other by a space.

The frequency count is multiplied by the number of instructions that are in the basic block, in order to weigh the count so that instructions in small basic blocks aren't counted as more important than instructions in large basic blocks.

The SimPoint program only processes lines that start with a "T". All other lines are ignored. Traditionally comments are indicated by starting a line with a "#" character. Some other BBV generation tools, such as PinPoints, generate lines beginning with letters other than "T" to indicate more information about the program being run. We do not generate these, as the SimPoint utility ignores them.

13.5. Implementation

Valgrind provides all of the information necessary to create BBV files. In the current implementation, all instructions are instrumented. This is slower (by approximately a factor of two) than a method that instruments at the basic block level, but there are some complications (especially with rep prefix detection) that make that method more difficult.

Valgrind actually provides instrumentation at a superblock level. A superblock has one entry point but unlike basic blocks can have multiple exit points. Once a branch occurs into the middle of a block, it is split into a new basic block. Because Valgrind cannot produce "true" basic blocks, the generated BBV vectors will be different than those generated by other tools. In practice this does not seem to affect the accuracy of the SimPoint results. We do internally force the `--vex-guest-chase=no` option to Valgrind which forces a more basic-block-like behavior.

When a superblock is run for the first time, it is instrumented with our BBV routine. A block info (bbInfo) structure is allocated which holds the various information and statistics for the block. A unique block ID is assigned to the block, and then the structure is placed into an ordered set. Then each native instruction in the block is instrumented to call an instruction counting routine with a pointer to the block info structure as an argument.

At run-time, our instruction counting routines are called once per native instruction. The relevant block info structure is accessed and the block count and total instruction count is updated. If the total instruction count overflows the interval size then we walk the ordered set, writing out the statistics for any block that was accessed in the interval, then resetting the block counters to zero.

On the x86 and amd64 architectures the counting code has extra code to handle rep-prefixed string instructions. This is because actual hardware counts a rep-prefixed instruction as one instruction, while a naive Valgrind implementation would count it as many (possibly hundreds, thousands or even millions) of instructions. We handle rep-prefixed instructions specially, in order to make the results match those obtained with hardware performance counters.

BBV also counts the fldcw instruction. This instruction is used on x86 machines in various ways; it is most commonly found when converting floating point values into integers. On Pentium 4 systems the retired instruction performance counter counts this instruction as two instructions (all other known processors only count it as one). This can affect results when using SimPoint on Pentium 4 systems. We provide the fldcw count so that users can evaluate whether it will impact their results enough to avoid using Pentium 4 machines for their experiments. It would be possible to add an option to this tool that mimics the double-counting so that the generated BBV files would be usable for experiments using hardware performance counters on Pentium 4 systems.

13.6. Threaded Executable Support

BBV supports threaded programs. When a program has multiple threads, an additional basic block vector file is created for each thread (each additional file is the specified filename with the thread number appended at the end).

There is no official method of using SimPoint with threaded workloads. The most common method is to run SimPoint on each thread's results independently, and use some method of deterministic execution to try to match the original workload. This should be possible with the current BBV.

13.7. Validation

BBV has been tested on x86, amd64, and ppc32 platforms. An earlier version of BBV was tested in detail using hardware performance counters, this work is described in a paper from the HiPEAC'08 conference, "Using Dynamic Binary Instrumentation to Generate Multi-Platform SimPoints: Methodology and Accuracy" by V.M. Weaver and S.A. McKee.

13.8. Performance

Using this program slows down execution by roughly a factor of 40 over native execution. This varies depending on the machine used and the benchmark being run. On the SPEC CPU 2000 benchmarks running on a 3.4GHz Pentium D processor, the slowdown ranges from 24x (mcf) to 340x (vortex.2).

Valgrind FAQ

Release 3.25.1 20 May 2025

Copyright © 2000-2025 Valgrind Developers

Email: valgrind@valgrind.org

Table of Contents

Valgrind Frequently Asked Questions

1. Background

1.1. How do you pronounce "Valgrind"?

The "Val" as in the word "value". The "grind" is pronounced with a short 'i' -- ie. "grinned" (rhymes with "tinned") rather than "grined" (rhymes with "find").

Don't feel bad: almost everyone gets it wrong at first.

1.2. Where does the name "Valgrind" come from?

From Nordic mythology. Originally (before release) the project was named Heimdall, after the watchman of the Nordic gods. He could "see a hundred miles by day or night, hear the grass growing, see the wool growing on a sheep's back", etc. This would have been a great name, but it was already taken by a security package "Heimdal".

Keeping with the Nordic theme, Valgrind was chosen. Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard). Over this entrance there resides a wolf and over it there is the head of a boar and on it perches a huge eagle, whose eyes can see to the far regions of the nine worlds. Only those judged worthy by the guardians are allowed to pass through Valgrind. All others are refused entrance.

It's not short for "value grinder", although that's not a bad guess.

2. Compiling, installing and configuring

2.1. When building Valgrind, 'make' dies partway with an assertion failure, something like this:

```
% make: expand.c:489: allocated_variable_append:
        Assertion 'current_variable_set_list->next != 0' failed.
```

It's probably a bug in 'make'. Some, but not all, instances of version 3.79.1 have this bug, see this. Try upgrading to a more recent version of 'make'. Alternatively, we have heard that unsetting the CFLAGS environment variable avoids the problem.

2.2. When building Valgrind, 'make' fails with this:

```
/usr/bin/ld: cannot find -lc
collect2: ld returned 1 exit status
```

You need to install the glibc-static-devel package.

3. Valgrind aborts unexpectedly

3.1. Programs run OK on Valgrind, but at exit produce a bunch of errors involving `__libc_freeres` and then die with a segmentation fault.

When the program exits, Valgrind runs the procedure `__libc_freeres` in glibc. This is a hook for memory debuggers, so they can ask glibc to free up any memory it has used. Doing that is needed to ensure that Valgrind doesn't incorrectly report space leaks in glibc.

The problem is that running `__libc_freeres` in older glibc versions causes this crash.

Workaround for 1.1.X and later versions of Valgrind: use the `--run-libc-freeres=no` option. You may then get space leak reports for glibc allocations (please don't report these to the glibc people, since they are not real leaks), but at least the program runs.

3.2. My (buggy) program dies like this:

```
valgrind: m_mallocfree.c:248 (get_bszB_as_is): Assertion 'bszB_lo == bszB_hi' failed.
```

or like this:

```
valgrind: m_mallocfree.c:442 (mk_inuse_bszB): Assertion 'bszB != 0' failed.
```

or otherwise aborts or crashes in m_mallocfree.c.

If Memcheck (the memory checker) shows any invalid reads, invalid writes or invalid frees in your program, the above may happen. Reason is that your program may trash Valgrind's low-level memory manager, which then dies with the above assertion, or something similar. The cure is to fix your program so that it doesn't do any illegal memory accesses. The above failure will hopefully go away after that.

3.3. My program dies, printing a message like this along the way:

```
vex x86->IR: unhandled instruction bytes: 0x66 0xF 0x2E 0x5
```

One possibility is that your program has a bug and erroneously jumps to a non-code address, in which case you'll get a SIGILL signal. Memcheck may issue a warning just before this happens, but it might not if the jump happens to land in addressable memory.

Another possibility is that Valgrind does not handle the instruction. If you are using an older Valgrind, a newer version might handle the instruction. However, all instruction sets have some obscure, rarely used instructions. Also, on amd64 there are an almost limitless number of combinations of redundant instruction prefixes, many of them undocumented but accepted by CPUs. So Valgrind will still have decoding failures from time to time. If this happens, please file a bug report.

3.4. I tried running a Java program (or another program that uses a just-in-time compiler) under Valgrind but something went wrong. Does Valgrind handle such programs?

Valgrind can handle dynamically generated code, so long as none of the generated code is later overwritten by other generated code. If this happens, though, things will go wrong as Valgrind will continue running its translations of the old code (this is true on x86 and amd64, on PowerPC there are explicit cache flush instructions which Valgrind detects and honours). You should try running with `--smc-check=all` in this case. Valgrind will run much more slowly, but should detect the use of the out-of-date code.

Alternatively, if you have the source code to the JIT compiler you can insert calls to the `VALGRIND_DISCARD_TRANSLATIONS` client request to mark out-of-date code, saving you from using `--smc-check=all`.

Apart from this, in theory Valgrind can run any Java program just fine, even those that use JNI and are partially implemented in other languages like C and C++. In practice, Java implementations tend to do nasty things that most programs do not, and Valgrind sometimes falls over these corner cases.

If your Java programs do not run under Valgrind, even with `--smc-check=all`, please file a bug report and hopefully we'll be able to fix the problem.

4. Valgrind behaves unexpectedly

4.1. My program uses the C++ STL and string classes. Valgrind reports 'still reachable' memory leaks involving these classes at the exit of the program, but there should be none.

First of all: relax, it's probably not a bug, but a feature. Many implementations of the C++ standard libraries use their own memory pool allocators. Memory for quite a number of destructed objects is not immediately freed and given back to the OS, but kept in the pool(s) for later re-use. The fact that the pools are not freed at the exit of the program cause Valgrind to report this memory as still reachable. The behaviour not to free pools at the exit could be called a bug of the library though.

Using GCC, you can force the STL to use malloc and to free memory as soon as possible by globally disabling memory caching. Beware! Doing so will probably slow down your program, sometimes drastically.

- With GCC 2.91, 2.95, 3.0 and 3.1, compile all source using the STL with `-D__USE_MALLOC`. Beware! This was removed from GCC starting with version 3.3.
- With GCC 3.2.2 and later, you should export the environment variable `GLIBCPP_FORCE_NEW` before running your program.

- With GCC 3.4 and later, that variable has changed name to `GLIBCXX_FORCE_NEW`.

There are other ways to disable memory pooling: using the `malloc_alloc` template with your objects (not portable, but should work for GCC) or even writing your own memory allocators. But all this goes beyond the scope of this FAQ. Start by reading http://gcc.gnu.org/onlinedocs/libstdc++/faq/index.html#4_4_leak if you absolutely want to do that. But beware: allocators belong to the more messy parts of the STL and people went to great lengths to make the STL portable across platforms. Chances are good that your solution will work on your platform, but not on others.

4.2. The stack traces given by Memcheck (or another tool) aren't helpful. How can I improve them?

If they're not long enough, use `--num-callers` to make them longer.

If they're not detailed enough, make sure you are compiling with `-g` to add debug information. And don't strip symbol tables (programs should be unstripped unless you run 'strip' on them; some libraries ship stripped).

Also, for leak reports involving shared objects, if the shared object is unloaded before the program terminates, Valgrind will discard the debug information and the error message will be full of `???` entries. If you use the option `--keep-debuginfo=yes`, then Valgrind will keep the debug information in order to show the stack traces, at the price of increased memory. An alternate workaround is to avoid calling `dlclose` on these shared objects.

Also, `-fomit-frame-pointer` and `-fstack-check` can make stack traces worse.

Some example sub-traces:

- With debug information and unstripped (best):

```
Invalid write of size 1
   at 0x80483BF: really (malloc1.c:20)
   by 0x8048370: main (malloc1.c:9)
```

- With no debug information, unstripped:

```
Invalid write of size 1
   at 0x80483BF: really (in /auto/homes/njn25/grind/head5/a.out)
   by 0x8048370: main (in /auto/homes/njn25/grind/head5/a.out)
```

- With no debug information, stripped:

```
Invalid write of size 1
   at 0x80483BF: (within /auto/homes/njn25/grind/head5/a.out)
   by 0x8048370: (within /auto/homes/njn25/grind/head5/a.out)
   by 0x42015703: __libc_start_main (in /lib/tls/libc-2.3.2.so)
   by 0x80482CC: (within /auto/homes/njn25/grind/head5/a.out)
```

- With debug information and -fomit-frame-pointer:

```
Invalid write of size 1
   at 0x80483C4: really (malloc1.c:20)
   by 0x42015703: __libc_start_main (in /lib/tls/libc-2.3.2.so)
   by 0x80482CC: ??? (start.S:81)
```

- A leak error message involving an unloaded shared object:

```
84 bytes in 1 blocks are possibly lost in loss record 488 of 713
   at 0x1B9036DA: operator new(unsigned) (vg_replace_malloc.c:132)
   by 0x1DB63EEB: ???
   by 0x1DB4B800: ???
   by 0x1D65E007: ???
   by 0x8049EE6: main (main.cpp:24)
```

4.3. The stack traces given by Memcheck (or another tool) seem to have the wrong function name in them. What's happening?

Occasionally Valgrind stack traces get the wrong function names. This is caused by glibc using aliases to effectively give one function two names. Most of the time Valgrind chooses a suitable name, but very occasionally it gets it wrong. Examples we know of are printing `bcmp` instead of `memcmp`, `index` instead of `strchr`, and `rindex` instead of `strrchr`.

4.4. My program crashes normally, but doesn't under Valgrind, or vice versa. What's happening?

When a program runs under Valgrind, its environment is slightly different to when it runs natively. For example, the memory layout is different, and the way that threads are scheduled is different.

Most of the time this doesn't make any difference, but it can, particularly if your program is buggy. For example, if your program crashes because it erroneously accesses memory that is unaddressable, it's possible that this memory will not be unaddressable when run under Valgrind. Alternatively, if your program has data races, these may not manifest under Valgrind.

There isn't anything you can do to change this, it's just the nature of the way Valgrind works that it cannot exactly replicate a native execution environment. In the case where your program crashes due to a memory error when run natively but not when run under Valgrind, in most cases Memcheck should identify the bad memory operation.

4.5. Memcheck doesn't report any errors and I know my program has errors.

There are two possible causes of this.

First, by default, Valgrind only traces the top-level process. So if your program spawns children, they won't be traced by Valgrind by default. Also, if your program is started by a shell script, Perl script, or something similar, Valgrind will trace the shell, or the Perl interpreter, or equivalent.

To trace child processes, use the `--trace-children=yes` option.

If you are tracing large trees of processes, it can be less disruptive to have the output sent over the network. Give Valgrind the option `--log-socket=127.0.0.1:12345` (if you want logging output sent to port `12345` on `localhost`). You can use the valgrind-listener program to listen on that port:

```
valgrind-listener 12345
```

Obviously you have to start the listener process first. See the manual for more details.

Second, if your program is statically linked, most Valgrind tools will only work well if they are able to replace certain functions, such as `malloc`, with their own versions. By default, statically linked `malloc functions` are not replaced. A key indicator of this is if Memcheck says:

```
All heap blocks were freed -- no leaks are possible
```

when you know your program calls `malloc`. The workaround is to use the option `--soname-synonyms=somalloc=NONE` or to avoid statically linking your program.

There will also be no replacement if you use an alternative `malloc library` such as tcmalloc, jemalloc, ... In such a case, the option `--soname-synonyms=somalloc=zzzz` (where zzzz is the soname of the alternative malloc library) will allow Valgrind to replace the functions.

4.6. Why doesn't Memcheck find the array overruns in this program?

```
int static[5];

int main(void)
{
  int stack[5];

  static[5] = 0;
  stack [5] = 0;

  return 0;
}
```

Unfortunately, Memcheck doesn't do bounds checking on global or stack arrays. We'd like to, but it's just not possible to do in a reasonable way that fits with how Memcheck works. Sorry.

4.7. Why does Memcheck report many "Mismatched free() / delete / delete []" errors when my code is correct?

There are two possible causes of this.

First, check if you are using an optimized build of Google tcmalloc (part of Google perftools). This library uses a single alias for free/scalar delete/array delete as an unmeasurable micro-optimization. There is simply no way for Memcheck to tell which of these was originally used. There are a few possible workarounds.

- Build tcmalloc with "CPPFLAGS=-DTCMALLOC_NO_ALIASES" (best).
- Use a debug build of tcmalloc (debug builds turn off the alias micro-optimization).
- Do not link with tcmalloc for the builds that you use for Memcheck testing.

Second, if you are replacing operator new or operator delete make sure that the compiler does not perform optimizations such as inlining on calls to these functions. Such optimizations can prevent Memcheck from correctly identifying the allocator or deallocator that is being used.

The following two code snippets show how you can do this with GCC and LLVM (clang).

```
  // GCC
  void operator delete(void*) noexcept __attribute__((__externally_visible__));
```

```
  // LLVM (clang)
  __attribute__((__visibility__("default"))) void operator delete(void*) noexcept;
```

If all else fails, you might have to use "--show-mismatched-frees=no"

5. Miscellaneous

5.1. I tried writing a suppression but it didn't work. Can you write my suppression for me?

Yes! Use the `--gen-suppressions=yes` feature to spit out suppressions automatically for you. You can then edit them if you like, eg. combining similar automatically generated suppressions using wildcards like `'*'`.

If you really want to write suppressions by hand, read the manual carefully. Note particularly that C++ function names must be mangled (that is, not demangled).

5.2. With Memcheck's memory leak detector, what's the difference between "definitely lost", "indirectly lost", "possibly lost", "still reachable", and "suppressed"?

The details are in the Memcheck section of the user manual.

In short:

- "definitely lost" means your program is leaking memory -- fix those leaks!
- "indirectly lost" means your program is leaking memory in a pointer-based structure. (E.g. if the root node of a binary tree is "definitely lost", all the children will be "indirectly lost".) If you fix the "definitely lost" leaks, the "indirectly lost" leaks should go away.
- "possibly lost" means your program is leaking memory, unless you're doing unusual things with pointers that could cause them to point into the middle of an allocated block; see the user manual for some possible causes. Use `--show-possibly-lost=no` if you don't want to see these reports.
- "still reachable" means your program is probably ok -- it didn't free some memory it could have. This is quite common and often reasonable. Don't use `--show-reachable=yes` if you don't want to see these reports.
- "suppressed" means that a leak error has been suppressed. There are some suppressions in the default suppression files. You can ignore suppressed errors.

5.3. Memcheck's uninitialised value errors are hard to track down, because they are often reported some time after they are caused. Could Memcheck record a trail of operations to better link the cause to the effect? Or maybe just eagerly report any copies of uninitialised memory values?

Prior to version 3.4.0, the answer was "we don't know how to do it without huge performance penalties". As of 3.4.0, try using the `--track-origins=yes` option. It will run slower than usual, but will give you extra information about the origin of uninitialised values.

Or if you want to do it the old fashioned way, you can use the client request `VALGRIND_CHECK_VALUE_IS_DEFINED` to help track these errors down -- work backwards from the point where the uninitialised error occurs, checking suspect values until you find the cause. This requires editing, compiling and re-running your program multiple times, which is a pain, but still easier than debugging the problem without Memcheck's help.

As for eager reporting of copies of uninitialised memory values, this has been suggested multiple times. Unfortunately, almost all programs legitimately copy uninitialised memory values around (because compilers pad structs to preserve alignment) and eager checking leads to hundreds of false positives. Therefore Memcheck does not support eager checking at this time.

5.4. I'm developing a Qt application and I get huge numbers of "Conditional jump" errors. Is there anything that I can do about it?

Yes, there is a workaround. Here is an example error:

```
Conditional jump or move depends on uninitialised value(s)
   at 0x1051C39B: ???
   by 0x12657AA7: ???
```

Qt Regular Expressions are built on the pcre2 library. pcre2 uses JITting which means that the errors cannot be suppressed (no function name). However, Qt provides a mechanism to turn off the use of JITting. To do so, use the following environment variable: `export QT_ENABLE_REGEXP_JIT=0`

5.5. Is it possible to attach Valgrind to a program that is already running?

No. The environment that Valgrind provides for running programs is significantly different to that for normal programs, e.g. due to different layout of memory. Therefore Valgrind has to have full control from the very start.

It is possible to achieve something like this by running your program without any instrumentation (which involves a slow-down of about 5x, less than that of most tools), and then adding instrumentation once you get to a point of interest. Support for this must be provided by the tool, however, and Callgrind is the only tool that currently has such support. See the instructions on the `callgrind_control` program for details.

6. How To Get Further Assistance

6.1. Where can I get more help?

Read the appropriate section(s) of the Valgrind Documentation.

Search the valgrind-users mailing list archives, using the group name `gmane.comp.debugging.valgrind`.

If you think an answer in this FAQ is incomplete or inaccurate, please e-mail valgrind@valgrind.org.

If you have tried all of these things and are still stuck, you can try mailing the valgrind-users mailing list. Note that an email has a better change of being answered usefully if it is clearly written. Also remember that, despite the fact that most of the community are very helpful and responsive to emailed questions, you are probably requesting help from unpaid volunteers, so you have no guarantee of receiving an answer.

Valgrind Technical Documentation

Release 3.25.1 20 May 2025

Copyright © 2000-2025 Valgrind Developers

Email: valgrind@valgrind.org

Table of Contents

1. The Design and Implementation of Valgrind

A number of academic publications nicely describe many aspects of Valgrind's design and implementation. Online copies of all of them, and others, are available on the Valgrind publications page.

The following paper gives a good overview of Valgrind, and explains how it differs from other dynamic binary instrumentation frameworks such as Pin and DynamoRIO.

- **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Nicholas Nethercote and Julian Seward. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.**

The following two papers together give a comprehensive description of how most of Memcheck works. The first paper describes in detail how Memcheck's undefined value error detection (a.k.a. V bits) works. The second paper describes in detail how Memcheck's shadow memory is implemented, and compares it to other alternative approaches.

- **Using Valgrind to detect undefined value errors with bit-precision. Julian Seward and Nicholas Nethercote. Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.**

  How to Shadow Every Byte of Memory Used by a Program. Nicholas Nethercote and Julian Seward. Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007), San Diego, California, USA, June 2007.

The following paper describes Callgrind.

- **A Tool Suite for Simulation Based Analysis of Memory Access Behavior. Josef Weidendorfer, Markus Kowarschik and Carsten Trinitis. Proceedings of the 4th International Conference on Computational Science (ICCS 2004), Krakow, Poland, June 2004.**

The following dissertation describes Valgrind in some detail (many of these details are now out-of-date) as well as Cachegrind, Annelid and Redux. It also covers some underlying theory about dynamic binary analysis in general and what all these tools have in common.

- **Dynamic Binary Analysis and Instrumentation. Nicholas Nethercote.** PhD Dissertation, University of Cambridge, November 2004.

2. Writing a New Valgrind Tool

So you want to write a Valgrind tool? Here are some instructions that may help.

2.1. Introduction

The key idea behind Valgrind's architecture is the division between its *core* and *tools*.

The core provides the common low-level infrastructure to support program instrumentation, including the JIT compiler, low-level memory manager, signal handling and a thread scheduler. It also provides certain services that are useful to some but not all tools, such as support for error recording, and support for replacing heap allocation functions such as `malloc`.

But the core leaves certain operations undefined, which must be filled by tools. Most notably, tools define how program code should be instrumented. They can also call certain functions to indicate to the core that they would like to use certain services, or be notified when certain interesting events occur. But the core takes care of all the hard work.

2.2. Basics

2.2.1. How tools work

Tools must define various functions for instrumenting programs that are called by Valgrind's core. They are then linked against Valgrind's core to define a complete Valgrind tool which will be used when the `--tool` option is used to select it.

2.2.2. Getting the code

To write your own tool, you'll need the Valgrind source code. You'll need a clone from the git repository for the automake/autoconf build instructions to work. See the information about how to do clone from the repository at the Valgrind website.

2.2.3. Getting started

Valgrind uses GNU `automake` and `autoconf` for the creation of Makefiles and configuration. But don't worry, these instructions should be enough to get you started even if you know nothing about those tools.

In what follows, all filenames are relative to Valgrind's top-level directory `valgrind/`.

1. Choose a name for the tool, and a two-letter abbreviation that can be used as a short prefix. We'll use `foobar` and `fb` as an example.

2. Make three new directories `foobar/`, `foobar/docs/` and `foobar/tests/`.

3. Create an empty file `foobar/tests/Makefile.am`.

4. Copy `none/Makefile.am` into `foobar/`. Edit it by replacing all occurrences of the strings `"none"`, `"nl_"` and `"nl-"` with `"foobar"`, `"fb_"` and `"fb-"` respectively.

5. Copy `none/nl_main.c` into `foobar/`, renaming it as `fb_main.c`. Edit it by changing the `details` lines in `nl_pre_clo_init` to something appropriate for the tool. These fields are used in the startup message, except

for `bug_reports_to` which is used if a tool assertion fails. Also, replace the string `"nl_"` throughout with `"fb_"` again.

6. Edit `Makefile.am`, adding the new directory `foobar` to the `TOOLS` or `EXP_TOOLS` variables.

7. Edit `configure.ac`, adding `foobar/Makefile` and `foobar/tests/Makefile` to the `AC_OUTPUT` list.

8. Run:

```
autogen.sh
./configure --prefix=`pwd`/inst
make
make install
```

   It should automake, configure and compile without errors, putting copies of the tool in `foobar/` and `inst/lib/valgrind/`.

9. You can test it with a command like:

```
inst/bin/valgrind --tool=foobar date
```

   (almost any program should work; `date` is just an example). The output should be something like this:

```
==738== foobar-0.0.1, a foobarring tool.
==738== Copyright (C) 2002-2017, and GNU GPL'd, by J. Programmer.
==738== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==738== Command: date
==738==
Tue Nov 27 12:40:49 EST 2017
==738==
```

   The tool does nothing except run the program uninstrumented.

These steps don't have to be followed exactly -- you can choose different names for your source files, and use a different `--prefix` for `./configure`.

Now that we've setup, built and tested the simplest possible tool, onto the interesting stuff...

2.2.4. Writing the code

A tool must define at least these four functions:

```
pre_clo_init()
post_clo_init()
instrument()
fini()
```

The names can be different to the above, but these are the usual names. The first one is registered using the macro `VG_DETERMINE_INTERFACE_VERSION`. The last three are registered using the `VG_(basic_tool_funcs)` function.

In addition, if a tool wants to use some of the optional services provided by the core, it may have to define other functions and tell the core about them.

2.2.5. Initialisation

Most of the initialisation should be done in `pre_clo_init`. Only use `post_clo_init` if a tool provides command line options and must do some initialisation after option processing takes place (`"clo"` stands for "command line options").

First of all, various "details" need to be set for a tool, using the functions `VG_(details_*)`. Some are all compulsory, some aren't. Some are used when constructing the startup message, `detail_bug_reports_to` is used if `VG_(tool_panic)` is ever called, or a tool assertion fails. Others have other uses.

Second, various "needs" can be set for a tool, using the functions `VG_(needs_*)`. They are mostly booleans, and can be left untouched (they default to `False`). They determine whether a tool can do various things such as: record, report and suppress errors; process command line options; wrap system calls; record extra information about heap blocks; etc.

For example, if a tool wants the core's help in recording and reporting errors, it must call `VG_(needs_tool_errors)` and provide definitions of eight functions for comparing errors, printing out errors, reading suppressions from a suppressions file, etc. While writing these functions requires some work, it's much less than doing error handling from scratch because the core is doing most of the work.

Third, the tool can indicate which events in core it wants to be notified about, using the functions `VG_(track_*)`. These include things such as heap blocks being allocated, the stack pointer changing, a mutex being locked, etc. If a tool wants to know about this, it should provide a pointer to a function, which will be called when that event happens.

For example, if the tool want to be notified when a new heap block is allocated, it should call `VG_(track_new_mem_heap)` with an appropriate function pointer, and the assigned function will be called each time this happens.

More information about "details", "needs" and "trackable events" can be found in `include/pub_tool_tooliface.h`.

2.2.6. Instrumentation

`instrument` is the interesting one. It allows you to instrument *VEX IR*, which is Valgrind's RISC-like intermediate language. VEX IR is described in the comments of the header file `VEX/pub/libvex_ir.h`.

The easiest way to instrument VEX IR is to insert calls to C functions when interesting things happen. See the tool "Lackey" (`lackey/lk_main.c`) for a simple example of this, or Cachegrind (`cachegrind/cg_main.c`) for a more complex example.

2.2.7. Finalisation

This is where you can present the final results, such as a summary of the information collected. Any log files should be written out at this point.

2.2.8. Other Important Information

Please note that the core/tool split infrastructure is quite complex and not brilliantly documented. Here are some important points, but there are undoubtedly many others that I should note but haven't thought of.

The files `include/pub_tool_*.h` contain all the types, macros, functions, etc. that a tool should (hopefully) need, and are the only `.h` files a tool should need to `#include`. They have a reasonable amount of documentation in it that should hopefully be enough to get you going.

Note that you can't use anything from the C library (there are deep reasons for this, trust us). Valgrind provides an implementation of a reasonable subset of the C library, details of which are in `pub_tool_libc*.h`.

When writing a tool, in theory you shouldn't need to look at any of the code in Valgrind's core, but in practice it might be useful sometimes to help understand something.

The `include/pub_tool_basics.h` and `VEX/pub/libvex_basictypes.h` files have some basic types that are widely used.

Ultimately, the tools distributed (Memcheck, Cachegrind, Lackey, etc.) are probably the best documentation of all, for the moment.

The `VG_` macro is used heavily. This just prepends a longer string in front of names to avoid potential namespace clashes. It is defined in `include/pub_tool_basics.h`.

There are some assorted notes about various aspects of the implementation in `docs/internals/`. Much of it isn't that relevant to tool-writers, however.

2.3. Advanced Topics

Once a tool becomes more complicated, there are some extra things you may want/need to do.

2.3.1. Debugging Tips

Writing and debugging tools is not trivial. Here are some suggestions for solving common problems.

If you are getting segmentation faults in C functions used by your tool, the usual GDB command:

```
gdb <prog> core
```

usually gives the location of the segmentation fault.

If you want to debug C functions used by your tool, there are instructions on how to do so in the file `README_DEVELOPERS`.

If you are having problems with your VEX IR instrumentation, it's likely that GDB won't be able to help at all. In this case, Valgrind's `--trace-flags` option is invaluable for observing the results of instrumentation.

If you just want to know whether a program point has been reached, using the `OINK` macro (in `include/pub_tool_libcprint.h`) can be easier than using GDB.

The other debugging command line options can be useful too (run `valgrind --help-debug` for the list).

2.3.2. Suppressions

If your tool reports errors and you want to suppress some common ones, you can add suppressions to the suppression files. The relevant files are `*.supp`; the final suppression file is aggregated from these files by combining the relevant `.supp` files depending on the versions of linux, X and glibc on a system.

Suppression types have the form `tool_name:suppression_name`. The `tool_name` here is the name you specify for the tool during initialisation with `VG_(details_name)`.

2.3.3. Documentation

If you are feeling conscientious and want to write some documentation for your tool, please use XML as the rest of Valgrind does. The file `docs/README` has more details on getting the XML toolchain to work; this can be difficult, unfortunately.

To write the documentation, follow these steps (using `foobar` as the example tool name again):

1. The docs go in `foobar/docs/`, which you will have created when you started writing the tool.
2. Copy the XML documentation file for the tool Nulgrind from `none/docs/nl-manual.xml` to `foobar/docs/`, and rename it to `foobar/docs/fb-manual.xml`.

   Note: there is a tetex bug involving underscores in filenames, so don't use '_'.
3. Write the documentation. There are some helpful bits and pieces on using XML markup in `docs/xml/xml_help.txt`.
4. Include it in the User Manual by adding the relevant entry to `docs/xml/manual.xml`. Copy and edit an existing entry.
5. Include it in the man page by adding the relevant entry to `docs/xml/valgrind-manpage.xml`. Copy and edit an existing entry.
6. Validate `foobar/docs/fb-manual.xml` using the following command from within `docs/`:

   ```
   make valid
   ```

   You may get errors that look like this:

   ```
   ./xml/index.xml:5: element chapter: validity error : No declaration for
   attribute base of element chapter
   ```

   Ignore (only) these -- they're not important.

   Because the XML toolchain is fragile, it is important to ensure that `fb-manual.xml` won't break the documentation set build. Note that just because an XML file happily transforms to html does not necessarily mean the same holds true for pdf/ps.
7. You can (re-)generate the HTML docs while you are writing `fb-manual.xml` to help you see how it's looking. The generated files end up in `docs/html/`. Use the following command, within `docs/`:

   ```
   make html-docs
   ```
8. When you have finished, try to generate PDF and PostScript output to check all is well, from within `docs/`:

   ```
   make print-docs
   ```

   Check the output `.pdf` and `.ps` files in `docs/print/`.

   Note that the toolchain is even more fragile for the print docs, so don't feel too bad if you can't get it working.

2.3.4. Regression Tests

Valgrind has some support for regression tests. If you want to write regression tests for your tool:

1. The tests go in `foobar/tests/`, which you will have created when you started writing the tool.
2. Write `foobar/tests/Makefile.am`. Use `memcheck/tests/Makefile.am` as an example.

3. Write the tests, `.vgtest` test description files, `.stdout.exp` and `.stderr.exp` expected output files. (Note that Valgrind's output goes to stderr.) Some details on writing and running tests are given in the comments at the top of the testing script `tests/vg_regtest`.

4. Write a filter for stderr results `foobar/tests/filter_stderr`. It can call the existing filters in `tests/`. See `memcheck/tests/filter_stderr` for an example; in particular note the `$dir` trick that ensures the filter works correctly from any directory.

2.3.5. Profiling

Lots of profiling tools have trouble running Valgrind. For example, trying to use gprof is hopeless.

Probably the best way to profile a tool is with OProfile on Linux.

You can also use Cachegrind on it. Read `README_DEVELOPERS` for details on running Valgrind under Valgrind; it's a bit fragile but can usually be made to work.

2.3.6. Other Makefile Hackery

If you add any directories under `foobar/`, you will need to add an appropriate `Makefile.am` to it, and add a corresponding entry to the `AC_OUTPUT` list in `configure.ac`.

If you add any scripts to your tool (see Cachegrind for an example) you need to add them to the `bin_SCRIPTS` variable in `foobar/Makefile.am` and possible also to the `AC_OUTPUT` list in `configure.ac`.

2.3.7. The Core/tool Interface

The core/tool interface evolves over time, but it's pretty stable. We deliberately do not provide backward compatibility with old interfaces, because it is too difficult and too restrictive. We view this as a good thing -- if we had to be backward compatible with earlier versions, many improvements now in the system could not have been added.

Because tools are statically linked with the core, if a tool compiles successfully then it should be compatible with the core. We would not deliberately violate this property by, for example, changing the behaviour of a core function without changing its prototype.

2.4. Final Words

Writing a new Valgrind tool is not easy, but the tools you can write with Valgrind are among the most powerful programming tools there are. Happy programming!

3. Callgrind Format Specification

This chapter describes the Callgrind Format, Version 1.

The format description is meant for the user to be able to understand the file contents; but more important, it is given for authors of measurement or visualization tools to be able to write and read this format.

3.1. Overview

The profile data format is ASCII based. It is written by Callgrind, and it is upwards compatible to the format used by Cachegrind (ie. Cachegrind uses a subset). It can be read by callgrind_annotate and KCachegrind.

This chapter gives on overview of format features and examples. For detailed syntax, look at the format reference.

3.1.1. Basic Structure

To uniquely specify that a file is a callgrind profile, it should add "# callgrind format" as first line. This is optional but recommended for easy format detection.

Each file has a header part of an arbitrary number of lines of the format "key: value". After the header, lines specifying profile costs follow. Everywhere, comments on own lines starting with '#' are allowed. The header lines with keys "positions" and "events" define the meaning of cost lines in the second part of the file: the value of "positions" is a list of subpositions, and the value of "events" is a list of event type names. Cost lines consist of subpositions followed by 64-bit counters for the events, in the order specified by the "positions" and "events" header line.

The "events" header line is always required in contrast to the optional line for "positions", which defaults to "line", i.e. a line number of some source file. In addition, the second part of the file contains position specifications of the form "spec=name". "spec" can be e.g. "fn" for a function name or "fl" for a file name. Cost lines are always related to the function/file specifications given directly before.

3.1.2. Simple Example

The event names in the following example are quite arbitrary, and are not related to event names used by Callgrind. Especially, cycle counts matching real processors probably will never be generated by any Valgrind tools, as these are bound to simulations of simple machine models for acceptable slowdown. However, any profiling tool could use the format described in this chapter.

```
# callgrind format
events: Cycles Instructions Flops
fl=file.f
fn=main
15 90 14 2
16 20 12
```

The above example gives profile information for event types "Cycles", "Instructions", and "Flops". Thus, cost lines give the number of CPU cycles passed by, number of executed instructions, and number of floating point operations executed while running code corresponding to some source position. As there is no line specifying the value of "positions", it defaults to "line", which means that the first number of a cost line is always a line number.

Thus, the first cost line specifies that in line 15 of source file `file.f` there is code belonging to function `main`. While running, 90 CPU cycles passed by, and 2 of the 14 instructions executed were floating point operations. Similarly, the next line specifies that there were 12 instructions executed in the context of function `main` which can be related to line 16 in file `file.f`, taking 20 CPU cycles. If a cost line specifies less event counts than given in the "events" line, the rest is assumed to be zero. I.e. there was no floating point instruction executed relating to line 16.

Note that regular cost lines always give self (also called exclusive) cost of code at a given position. If you specify multiple cost lines for the same position, these will be summed up. On the other hand, in the example above there is no specification of how many times function `main` actually was called: profile data only contains sums.

3.1.3. Associations

The most important extension to the original format of Cachegrind is the ability to specify call relationship among functions. More generally, you specify associations among positions. For this, the second part of the file also can contain association specifications. These look similar to position specifications, but consist of two lines. For calls, the format looks like

```
calls=(Call Count) (Target position)
(Source position) (Inclusive cost of call)
```

The destination only specifies subpositions like line number. Therefore, to be able to specify a call to another function in another source file, you have to precede the above lines with a "cfn=" specification for the name of the called function, and optionally a "cfi=" specification if the function is in another source file ("cfl=" is an alternative specification for "cfi=" because of historical reasons, and both should be supported by format readers). The second line looks like a regular cost line with the difference that inclusive cost spent inside of the function call has to be specified.

Other associations are for example (conditional) jumps. See the reference below for details.

3.1.4. Extended Example

The following example shows 3 functions, `main`, `func1`, and `func2`. Function `main` calls `func1` once and `func2` 3 times. `func1` calls `func2` 2 times.

```
# callgrind format
events: Instructions

fl=file1.c
fn=main
16 20
cfn=func1
calls=1 50
16 400
cfi=file2.c
cfn=func2
calls=3 20
16 400

fn=func1
51 100
cfi=file2.c
cfn=func2
calls=2 20
51 300
```

```
fl=file2.c
fn=func2
20 700
```

One can see that in `main` only code from line 16 is executed where also the other functions are called. Inclusive cost of `main` is 820, which is the sum of self cost 20 and costs spent in the calls: 400 for the single call to `func1` and 400 as sum for the three calls to `func2`.

Function `func1` is located in `file1.c`, the same as `main`. Therefore, a "cfi=" specification for the call to `func1` is not needed. The function `func1` only consists of code at line 51 of `file1.c`, where `func2` is called.

3.1.5. Name Compression

With the introduction of association specifications like calls it is needed to specify the same function or same file name multiple times. As absolute filenames or symbol names in C++ can be quite long, it is advantageous to be able to specify integer IDs for position specifications. Here, the term "position" corresponds to a file name (source or object file) or function name.

To support name compression, a position specification can be not only of the format "spec=name", but also "spec=(ID) name" to specify a mapping of an integer ID to a name, and "spec=(ID)" to reference a previously defined ID mapping. There is a separate ID mapping for each position specification, i.e. you can use ID 1 for both a file name and a symbol name.

With string compression, the example from above looks like this:

```
# callgrind format
events: Instructions

fl=(1) file1.c
fn=(1) main
16 20
cfn=(2) func1
calls=1 50
16 400
cfi=(2) file2.c
cfn=(3) func2
calls=3 20
16 400

fn=(2)
51 100
cfi=(2)
cfn=(3)
calls=2 20
51 300

fl=(2)
fn=(3)
20 700
```

As position specifications carry no information themselves, but only change the meaning of subsequent cost lines or associations, they can appear everywhere in the file without any negative consequence. Especially, you can define name compression mappings directly after the header, and before any cost lines. Thus, the above example can also be written as

```
# callgrind format
events: Instructions

# define file ID mapping
fl=(1) file1.c
fl=(2) file2.c
# define function ID mapping
fn=(1) main
fn=(2) func1
fn=(3) func2

fl=(1)
fn=(1)
16 20
...
```

3.1.6. Subposition Compression

If a Callgrind data file should hold costs for each assembler instruction of a program, you specify subposition "instr" in the "positions:" header line, and each cost line has to include the address of some instruction. Addresses are allowed to have a size of 64 bits to support 64-bit architectures. Thus, repeating similar, long addresses for almost every line in the data file can enlarge the file size quite significantly, and motivates for subposition compression: instead of every cost line starting with a 16 character long address, one is allowed to specify relative addresses. This relative specification is not only allowed for instruction addresses, but also for line numbers; both addresses and line numbers are called "subpositions".

A relative subposition always is based on the corresponding subposition of the last cost line, and starts with a "+" to specify a positive difference, a "-" to specify a negative difference, or consists of "*" to specify the same subposition. Because absolute subpositions always are positive (ie. never prefixed by "-"), any relative specification is non-ambiguous; additionally, absolute and relative subposition specifications can be mixed freely. Assume the following example (subpositions can always be specified as hexadecimal numbers, beginning with "0x"):

```
# callgrind format
positions: instr line
events: ticks

fn=func
0x80001234 90 1
0x80001237 90 5
0x80001238 91 6
```

With subposition compression, this looks like

```
# callgrind format
positions: instr line
events: ticks

fn=func
0x80001234 90 1
+3 * 5
+1 +1 6
```

Remark: For assembler annotation to work, instruction addresses have to be corrected to correspond to addresses found in the original binary. I.e. for relocatable shared objects, often a load offset has to be subtracted.

3.1.7. Miscellaneous

3.1.7.1. Cost Summary Information

For the visualization to be able to show cost percentage, a sum of the cost of the full run has to be known. Usually, it is assumed that this is the sum of all cost lines in a file. But sometimes, this is not correct. Thus, you can specify a "summary:" line in the header giving the full cost for the profile run. An import filter may use this to show a progress bar while loading a large data file.

3.1.7.2. Long Names for Event Types and inherited Types

Event types for cost lines are specified in the "events:" line with an abbreviated name. For visualization, it makes sense to be able to specify some longer, more descriptive name. For an event type "Ir" which means "Instruction Fetches", this can be specified the header line

```
event: Ir : Instruction Fetches
events: Ir Dr
```

In this example, "Dr" itself has no long name associated. The order of "event:" lines and the "events:" line is of no importance. Additionally, inherited event types can be introduced for which no raw data is available, but which are calculated from given types. Suppose the last example, you could add

```
event: Sum = Ir + Dr
```

to specify an additional event type "Sum", which is calculated by adding costs for "Ir and "Dr".

3.2. Reference

3.2.1. Grammar

```
ProfileDataFile := FormatSpec? FormatVersion? Creator? PartData*
```

```
FormatSpec := "# callgrind format\n"
```

```
FormatVersion := "version: 1\n"
```

```
Creator := "creator:" NoNewLineChar* "\n"
```

```
PartData := (HeaderLine "\n")+ (BodyLine "\n")+
```

```
HeaderLine := (empty line)
  | ('#' NoNewLineChar*)
  | PartDetail
  | Description
  | EventSpecification
  | CostLineDef
```

```
PartDetail := TargetCommand | TargetID
```

```
TargetCommand := "cmd:" Space* NoNewLineChar*
```

```
TargetID := ("pid"|"thread"|"part") ":" Space* Number
```

```
Description := "desc:" Space* Name Space* ":" NoNewLineChar*
```

```
EventSpecification := "event:" Space* Name InheritedDef? LongNameDef?
```

```
InheritedDef := "=" InheritedExpr
```

```
InheritedExpr := Name
  | Number Space* ("*" Space*)? Name
  | InheritedExpr Space* "+" Space* InheritedExpr
```

```
LongNameDef := ":" NoNewLineChar*
```

```
CostLineDef := "events:" Space* Name (Space+ Name)*
  | "positions:" "instr"? (Space+ "line")?
```

```
BodyLine := (empty line)
  | ('#' NoNewLineChar*)
  | CostLine
  | PositionSpec
  | CallSpec
  | UncondJumpSpec
  | CondJumpSpec
```

```
CostLine := SubPositionList Costs?
```

```
SubPositionList := (SubPosition+ Space+)+
```

```
SubPosition := Number | "+" Number | "-" Number | "*"
```

```
Costs := (Number Space+)+
```

```
PositionSpec := Position "=" Space* PositionName
```

```
Position := CostPosition | CalledPosition
```

```
CostPosition := "ob" | "fl" | "fi" | "fe" | "fn"
```

```
CalledPosition := " "cob" | "cfi" | "cfl" | "cfn"
```

```
PositionName := ( "(" Number ")" )? (Space* NoNewLineChar* )?
```

```
CallSpec := CallLine "\n" CostLine
```

```
CallLine := "calls=" Space* Number Space+ SubPositionList
```

```
UncondJumpSpec := "jump=" Space* Number Space+ SubPositionList
```

```
CondJumpSpec := "jcnd=" Space* Number Space+ Number Space+ SubPositionList
```

```
Space := " " | "\t"
```

```
Number := HexNumber | (Digit)+
```

```
Digit := "0" | ... | "9"
```

```
HexNumber := "0x" (Digit | HexChar)+
```

```
HexChar := "a" | ... | "f" | "A" | ... | "F"
```

```
Name = Alpha (Digit | Alpha)*
```

```
Alpha = "a" | ... | "z" | "A" | ... | "Z"
```

```
NoNewLineChar := all characters without "\n"
```

A profile data file ("ProfileDataFile") starts with basic information such as a format marker, the version and creator information, and then has a list of parts, where each part has its own header and body. Parts typically are different threads and/or time spans/phases within a profiled application run.

Note that callgrind_annotate currently only supports profile data files with one part. Callgrind may produce multiple parts for one profile run, but defaults to one output file for each part.

3.2.2. Description of Header Lines

Basic information in the first lines of a profile data file:

- `# callgrind format` [Callgrind]

  This line specifies that the file is a callgrind profile, and it has to be the first line. It was added late to the format (with Valgrind 3.13) and is optional, as all readers also should work with older callgrind profiles not including this line. However, generation of this line is recommended to allow desktop environments and file managers to uniquely detect the format.

- `version: number` [Callgrind]

  This is used to distinguish future profile data formats. A major version of 0 or 1 is supposed to be upwards compatible with Cachegrind's format. It is optional; if not appearing, version 1 is assumed. Otherwise, it has to follow directly after the format specification (i.e. be the first line if the optional format specification is skipped).

- `creator: string` [Callgrind]

  This is an arbitrary string to denote the creator of this file. Optional.

The header for each part has an arbitrary number of lines of the format "key: value". Possible *key* values for the header are:

- `pid: process id` [Callgrind]

  Optional. This specifies the process ID of the supervised application for which this profile was generated.

- `cmd: program name + args` [Cachegrind]

  Optional. This specifies the full command line of the supervised application for which this profile was generated.

- `part: number` [Callgrind]

  Optional. This specifies a sequentially incremented number for each dump generated, starting at 1.

- `desc: type: value` [Cachegrind]

  This specifies various information for this dump. For some types, the semantic is defined, but any description type is allowed. Unknown types should be ignored.

  There are the types "I1 cache", "D1 cache", "LL cache", which specify parameters used for the cache simulator. These are the only types originally used by Cachegrind. Additionally, Callgrind uses the following types: "Timerange" gives a rough range of the basic block counter, for which the cost of this dump was collected. Type "Trigger" states the reason of why this trace was generated. E.g. program termination or forced interactive dump.

- `positions: [instr] [line]` [Callgrind]

  For cost lines, this defines the semantic of the first numbers. Any combination of "instr", "bb" and "line" is allowed, but has to be in this order which corresponds to position numbers at the start of the cost lines later in the file.

  If "instr" is specified, the position is the address of an instruction whose execution raised the events given later on the line. This address is relative to the offset of the binary/shared library file to not have to specify relocation info. For "line", the position is the line number of a source file, which is responsible for the events raised. Note that the mapping of "instr" and "line" positions are given by the debugging line information produced by the compiler.

  This header line is optional, defaulting to "positions: line" if not specified.

- `events: event type abbreviations` [Cachegrind]

  A list of short names of the event types logged in cost lines in this part of the profile data file. Arbitrary short names are allowed. The order given specifies the required order in cost lines. Thus, the first event type is the second or third number in a cost line, depending on the value of "positions". Required to appear for each header part exactly once.

- `summary: costs` [Callgrind]

  Optional. This header line specifies a summary cost, which should be equal or larger than a total over all self costs. It may be larger as the cost lines may not represent all cost of the program run.

- `totals: costs` [Cachegrind]

  Optional. Should appear at the end of the file (although looking like a header line). Must give the total of all cost lines, to allow for a consistency check.

3.2.3. Description of Body Lines

The regular body line is a cost line consisting of one or two position numbers (depending on "positions:" header line, see above) and an array of cost numbers. A position number either is a line numbers into a source file or an instruction address within binary code, with source/binary file names specified as position names (see below). The cost numbers get mapped to event types in the same order as specified in the "events:" header line. If less numbers than event types are given, the costs default to zero for the remaining event types.

Further, there exist lines `spec=position name`. A position name is an arbitrary string. If it starts with "(" and a digit, it's a string in compressed format. Otherwise it's the real position string. This allows for file and symbol names as position strings, as these never start with "(" + *digit*. The compressed format is either "(" *number* ")" *space position* or only "(" *number* ")". The first relates *position* to *number* in the context of the given format specification from this line to the end of the file; it makes the (*number*) an alias for *position*. Compressed format is always optional.

Position specifications allowed:

- `ob=` [Callgrind]

  The ELF object where the cost of next cost lines happens.

- `fl=` [Cachegrind]
- `fi=` [Cachegrind]
- `fe=` [Cachegrind]

  The source file including the code which is responsible for the cost of next cost lines. "fi="/"fe=" is used when the source file changes inside of a function, i.e. for inlined code.

- `fn=` [Cachegrind]

  The name of the function where the cost of next cost lines happens.

- `cob=` [Callgrind]

  The ELF object of the target of the next call cost lines.

- `cfi=` [Callgrind]

  The source file including the code of the target of the next call cost lines.

- `cfl=` [Callgrind]

  Alternative spelling for `cfi=` specification (because of historical reasons).

- `cfn=` [Callgrind]

  The name of the target function of the next call cost lines.

The last type of body line provides specific costs not just related to one position as regular cost lines. It starts with specific strings similar to position name specifications.

- `calls=count target-position` [Callgrind]

  Call executed "count" times to "target-position". After a "calls=" line there MUST be a cost line. This provides the source position of the call and the cost spent in the called function in total.

- `jump=count target-position` [Callgrind]

  Unconditional jump, executed "count" times, to "target-position".

- `jcnd=exe-count jump-count target-position` [Callgrind]

  Conditional jump, executed "exe-count" times with "jump-count" jumps happening (rest is fall-through) to "target-position".

Valgrind Distribution Documents

Release 3.25.1 20 May 2025

Copyright © 2000-2025 Valgrind Developers

Email: valgrind@valgrind.org

Table of Contents

1. AUTHORS

Julian Seward was the original founder, designer and author of
Valgrind, created the dynamic translation frameworks, wrote Memcheck,
the 3.X versions of Helgrind, SGCheck, DHAT, and did lots of other
things.

Nicholas Nethercote did the core/tool generalisation, wrote
Cachegrind and Massif, and tons of other stuff.

Tom Hughes did a vast number of bug fixes, helped out with support for
more recent Linux/glibc versions, set up the present build system, and has
helped out with test and build machines.

Jeremy Fitzhardinge wrote Helgrind (in the 2.X line) and totally
overhauled low-level syscall/signal and address space layout stuff,
among many other things.

Josef Weidendorfer wrote and maintains Callgrind and the associated
KCachegrind GUI.

Paul Mackerras did a lot of the initial per-architecture factoring
that forms the basis of the 3.0 line and was also seen in 2.4.0.
He also did UCode-based dynamic translation support for PowerPC, and
created a set of ppc-linux derivatives of the 2.X release line.

Greg Parker wrote the Mac OS X port.

Dirk Mueller contributed the malloc/free mismatch checking
and other bits and pieces, and acts as our KDE liaison.

Robert Walsh added file descriptor leakage checking, new library
interception machinery, support for client allocation pools, and minor
other tweakage.

Bart Van Assche wrote and maintains DRD.

Cerion Armour-Brown worked on PowerPC instruction set support in the
Vex dynamic-translation framework. Maynard Johnson improved the
Power6 support.

Kirill Batuzov and Dmitry Zhurikhin did the NEON instruction set
support for ARM. Donna Robinson did the v6 media instruction support.

Donna Robinson created and maintains the very excellent
http://www.valgrind.org.

Vince Weaver wrote and maintains BBV.

Frederic Gobry helped with autoconf and automake.

Daniel Berlin modified readelf's dwarf2 source line reader, written by Nick Clifton, for use in Valgrind.

Michael Matz and Simon Hausmann modified the GNU binutils demangler(s) for use in Valgrind.

David Woodhouse has helped out with test and build machines over the course of many releases.

Florian Krohm and Christian Borntraeger wrote the initial S390X/Linux port. Andreas Arnez is the current maintainer and developer of it. Florian improved and ruggedised the regression test system during 2011.

Philippe Waroquiers wrote and maintains the embedded GDB server. He also made a bunch of performance and memory-reduction fixes across diverse parts of the system.

Carl Love and Maynard Johnson contributed IBM Power6 and Power7 support, and generally deal with ppc{32,64}-linux issues.

Petar Jovanovic and Dejan Jevtic wrote and maintain the mips32-linux port.

Dragos Tatulea modified the arm-android port so it also works on x86-android.

Jakub Jelinek helped out extensively with the AVX and AVX2 support.

Mark Wielaard fixed a bunch of bugs and acts as our Fedora/RHEL liaison.

Assad Hashmi contributed support for AArch64 v8.1 and later.

Maran Pakkirisamy implemented support for decimal floating point on s390.

Rhys Kidd updated and maintains the macOS port.

Paul Floyd maintains the FreeBSD port and occasionally fixes Solaris and macOS issues.

Many, many people sent bug reports, patches, and helpful feedback.

Development of Valgrind was supported in part by the Tri-Lab Partners (Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratories) of the U.S. Department of Energy's Advanced Simulation & Computing (ASC) Program.

2. NEWS

Release 3.25.1 (20 May 2025)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This point release contains only bug fixes.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved in this point release.

503098 Incorrect NAN-boxing for float registers in RISC-V
503641 close_range syscalls started failing with 3.25.0
503914 mount syscall param filesystemtype may be NULL
504177 FILE DESCRIPTORS banner shows when closing some inherited fds
504265 FreeBSD: missing syscall wrappers for fchroot and setcred
504466 Double close causes SEGV

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

Release 3.25.0 (25 Apr 2025)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, RISCV64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD, AMD64/FreeBSD and ARM64/FreeBSD There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* The valgrind gdbserver now supports the GDB remote protocol packet
  'x addr,len' (available in GDB release >= 16).
  The x packet can reduce the time taken by GDB to read memory from valgrind.

* Valgrind now supports zstd compressed debug sections.

* The Linux Test Project (ltp) is integrated in the testsuite try
  'make ltpchecks' (this will take a while and will point out various
  missing syscalls and valgrind crashes!)

* ================== PLATFORM CHANGES =================

* Added RISCV64 support for Linux. Specifically for the RV64GC
  instruction set.

* Numerous bug fixes for Illumos, in particular fixed a Valgrind crash

whenever a signal handler was called.

* On FreeBSD, a change to the libc code that runs atexit handlers was
causing Helgrind to produce an extra error about exiting threads
still holding locks for. This applied to every multithreaded application.
The extra error is now filtered out. A syscall wrapper had been added
for getrlimitusage.

* On Linux various new syscalls are supported (landlock*, io_pgetevents,
open_tree, move_mount, fsopen, fsconfig, fsmount, fspick, userfaultfd).

* s390x has support for various new instructions (BPP, BPRP, PPA and NIAI).

* ==================== TOOL CHANGES ===================

* The --track-fds=yes and --track-fds=all options now treat all
inherited file descriptors the same as 0, 1, 2 (stdin/out/err).
And when the stdin/out/err descriptors are reassigned they are
now treated as normal (non-inherited) file descriptors.

* A new option --modify-fds=high can be used together with
--track-fds=yes to create new file descriptors with the highest
possible number (and then decreasing) instead of always using the
lowest possible number (which is required by POSIX). This will help
catch issues where a file descriptor number might normally be reused
between a close and another open call.

* Helgrind:
There is a change to warnings about calls to pthread_cond_signal and
pthread_cond_broadcast when the associated mutex is unlocked. Previously
Helgrind would always warn about this. Now this error is controlled by
a command line option, --check-cond-signal-mutex=yes|no. The default is
no. This change has been made because some C and C++ standard libraries
use pthread_cond_signal/pthread_cond_broadcast in this way. Users are
obliged to use suppressions if they wish to avoid this noise.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

290061 pie elf always loaded at 0x108000
396415 Valgrind is not looking up $ORIGIN rpath of shebang programs
420682 io_pgetevents is not supported
468575 Add support for RISC-V
469782 Valgrind does not support zstd-compressed debug sections
487296 --track-fds=yes and --track-fds=all report erroneous information
when fds 0, 1, or 2 are used as non-std
489913 WARNING: unhandled amd64-linux syscall: 444 (landlock_create_ruleset)
493433 Add --modify-fds=[no|high] option

494246 syscall fsopen not wrapped
494327 Crash when running Helgrind built with #define TRACE_PTH_FNS 1
494337 All threaded applications cause still holding lock errors
495488 Add FreeBSD getrlimitusage syscall wrapper
495816 s390x: Fix disassembler segfault for C[G]RT and CL[G]RT
495817 s390x: Disassembly to match objdump -d output
496370 Illumos: signal handling is broken
496571 False positive for null key passed to bpf_map_get_next_key syscall.
496950 s390x: Fix hardware capabilities and EmFail codes
497130 Recognize new DWARF5 DW_LANG constants
497455 Update drd/scripts/download-and-build-gcc
497723 Enabling Ada demangling breaks callgrind differentiation between
    overloaded functions and procedures
498037 s390x: Add disassembly checker
498143 False positive on EVIOCGRAB ioctl
498317 FdBadUse is not a valid CoreError type in a suppression
    even though it's generated by --gen-suppressions=yes
498421 s390x: support BPP, BPRP and NIAI insns
498422 s390x: Fix VLRL and VSTRL insns
498492 none/tests/amd64/lzcnt64 crashes on FreeBSD compiled with clang
498629 s390x: Fix S[L]HHHR and S[L]HHLR insns
498632 s390x: Fix LNGFR insn
498942 s390x: Rework s390_disasm interface
499183 FreeBSD: differences in avx-vmovq output
499212 mmap() with MAP_ALIGNED() returns unaligned pointer
501119 memcheck/tests/pointer-trace fails when run on NFS filesystem
501194 Fix ML_(check_macho_and_get_rw_loads) so that it is correct for
    any number of segment commands
501348 glibc built with -march=x86-64-v3 does not work due to ld.so memcmp
501479 Illumos DRD pthread_mutex_init wrapper errors
501365 syscall userfaultfd not wrapped
501846 Add x86 Linux shm wrappers
501850 FreeBSD syscall arguments 7 and 8 incorrect.
501893 Missing suppression for __wcscat_avx2 (strcat-strlen-avx2.h.S:68)?
502126 glibc 2.41 extra syscall_cancel frames
502288 s390x: Memcheck false positives with NNPA last tensor dimension
502324 s390x: Memcheck false positives with TMxx and TM/TMY
502679 Use LTP for testing valgrind
502871 Make Helgrind "pthread_cond_{signal,broadcast}: dubious: associated
    lock is not held by any thread" optional

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

(3.25.0.RC1: 18 Apr 2025)
(3.25.0.RC2: 23 Apr 2025)

Release 3.24.0 (31 Oct 2024)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,
PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,
MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD, AMD64/FreeBSD
and ARM64/FreeBSD There is also preliminary support for X86/macOS 10.13,
AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* Bad file descriptor usage now generates a real error with
  --track-fds=yes that is suppressible and shows up in the xml output
  with full execution backtrace. The warnings shown without using the
  option are deprecated and will be removed in a future valgrind
  version.

* Ada name demangling is now supported in error messages.

* ================== PLATFORM CHANGES =================

* S390X added support for the DFLTCC instruction provided by the
  deflate-conversion facility (z15/arch13).

* S390X added support for the instructions provided by the MSA facility
  and MSA extensions 1-9.

* ==================== TOOL CHANGES ===================

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

202770 open fd at exit --log-socket=127.0.0.1:1500 with --track-fds=yes
276780 An instruction in fftw (Fast Fourier Transform) is unhandled by
        valgrind: vex x86->IR: unhandled instruction bytes:
        0x66 0xF 0x3A 0x2
311655 --log-file=FILE leads to apparent fd leak
317127 Fedora18/x86_64 --sanity-level=3 : aspacem segment mismatch
337388 fcntl works on Valgrind's own file descriptors
377966 arm64 unhandled instruction dc zva392146 aarch64: unhandled
        instruction 0xD5380001 (MRS rT, midr_el1)
391148 Unhandled AVX instruction vmovq %xmm9,%xmm1
392146 aarch64: unhandled instruction 0xD5380001 (MRS rT, midr_el1)
412377 SIGILL on cache flushes on arm64
417572 vex amd64->IR: unhandled instruction bytes: 0xC5 0x79 0xD6 0xED 0xC5
440180 s390x: Failed assertion in disassembler
444781 MIPS: wrong syscall numbers used
447989 Support Armv8.2 SHA-512 instructions
445235 Java/Ada/D demangling is probably broken
453044 gbserver_tests failures in aarch64
479661 Valgrind leaks file descriptors
486180 [Valgrind][MIPS] 'VexGuestArchState' has no member named
        'guest_IP_AT_SYSCALL'

486293 memccpy false positives
486569 linux inotify_init syscall wrapper missing POST entry in syscall_table
487439 SIGILL in JDK11, JDK17
487993 Alignment error when using Eigen with Valgrind and -m32
488026 Use of `sizeof` instead of `strlen
488379 --track-fds=yes errors that cannot be suppressed with --xml-file=
488441 Add tests for --track-fds=yes --xml=yes and fd suppression tests
489040 massif trace change to show the location increasing the stack
489088 Valgrind throws unhandled instruction bytes: 0xC5 0x79 0xD6 0xE0 0xC5
489338 arm64: Instruction fcvtas should round 322.5 to 323, but result is 322.
489676 vgdb handle EINTR and EAGAIN more consistently
490651 Stop using -flto-partition=one
491394 (vgModuleLocal_addDiCfSI): Assertion 'di->fsm.have_rx_map &&
        di->fsm.rw_map_count' failed
492210 False positive on x86/amd64 with ZF taken directly from addition
492214 statx(fd, NULL, AT_EMPTY_PATH) is supported since Linux 6.11
        but not supported in valgrind
492422 Please support DRM_IOCTL_SYNCOBJ_HANDLE_TO_FD
492663 Valgrind ignores debug info for some binaries
493418 Add bad fd usage errors for --track-fds in ML_(fd_allowed)
493454 Missing FUSE_COMPATIBLE_MAY_BLOCK markers
493507 direct readlink syscall from PRE handler is incompatible with
        FUSE_COMPATIBLE_MAY_BLOCK
493959 s390x: Fix regtest failure for none/tests/s390x/op00
493970 s390x: Store/restore FPC upon helper call causes slowdown
494218 Remove FREEBSD_VERS from configure and build
494252 s390x: incorrect disassembly for LOCHI and friends
494960 Fixes and tweaks for gsl19test
495278 PowerPC instruction dcbf should allow the L field values of 4, 6 on
        ISA 3.0 and earlier, just ignore the value
495469 aligned_alloc and posix_memalign missing MALLOC_TRACE with returned
        pointer
495470 s390x: 3.24.0.RC1 missing file and regtest failure
n-i-bz Improve messages for sigaltstack errors, use specific
        stack_t member names

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

(3.24.0.RC1: 27 Oct 2024)

Release 3.23.0 (26 Apr 2024)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD, AMD64/FreeBSD and ARM64/FreeBSD There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

* --track-fds=yes will now also warn about double closing of file
  descriptors. Printing the context where the file descriptor was
  originally opened and where it was previously closed.

* --track-fds=yes also produces "real" errors now which can be
  suppressed and work with --error-exitcode. When combined with
  --xml the xml-output now also includes FdBadClose and FdNotClosed
  error kinds (see docs/internals/xml-output-protocol5.txt).

* The option --show-error-list=no|yes now accepts a new value all.
  This indicates to also print the suppressed errors.
  This is useful to analyse which errors are suppressed by which
  suppression entries.
  The valgrind monitor command 'v.info all_errors' similarly now
  accepts a new optional argument 'also_suppressed' to show
  all errors including the suppressed errors.

* ================== PLATFORM CHANGES =================

* Added ARM64 support for FreeBSD.

* ARM64 now supports dotprod instructions (sdot/udot).

* AMD64 better supports code build with -march=x86-64-v3.
  fused-multiple-add instructions (fma) are now emulated more
  accurately. And memcheck now handles __builtin_strcmp using 128/256
  bit vectors with sse4.1, avx/avx2.

* S390X added support for NNPA (neural network processing assist)
  facility vector instructions VCNF, VCLFNH, VCFN, VCLFNL, VCRNF and
  NNPA (z16/arch14).

* X86 recognizes new binutils-2.42 nop patterns.

* ==================== TOOL CHANGES ===================

* The none tool now also supports xml output.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

283429 ARM leak checking needs CLEAR_CALLER_SAVED_REGS
281059 Cannot connect to Oracle using valgrind
328563 make track-fds support xml output
362680 --error-exitcode not honored when file descriptor leaks are found
369723 __builtin_longjmp not supported in clang/llvm on Android arm64 target

390269 unhandled amd64-darwin syscall: unix:464 (openat_nocancel)
401284 False positive "Source and destination overlap in strncat"
428364 Signals inside io_uring_enter not handled
437790 valgrind reports "Conditional jump or move depends on uninitialised value" in memchr of macOS 10.12-10.15
460616 disInstr(arm64): unhandled instruction 0x4E819402 (dotprod/ASIMDDP)
463458 memcheck/tests/vcpu_fnfns fails when glibc is built for x86-64-v3
463463 none/tests/amd64/fma fails when executed on a x86-64-v3 system
466762 Add redirs for C23 free_sized() and free_aligned_sized()
466884 Missing writev uninit padding suppression for _XSend
471036 disInstr_AMD64: disInstr miscalculated next %rip on RORX imm8, m32/64, r32/6
471222 support tracking of file descriptors being double closed
474160 If errors-for-leak-kinds is specified, exit-on-first-error should only exit on one of the listed errors.
475498 Add reallocarray wrapper
476025 Vbit expected test results for Iop_CmpGT64Ux2 are wrong
476320 Build failure with GCC
476331 clean up generated/distributed filter scripts
476535 Difference in allocation size for massif/tests/overloaded-new between clang++/libc++ and g++/libstdc++
476548 valgrind 3.22.0 fails on assertion when loading debuginfo file produced by mold
476708 valgrind-monitor.py regular expressions should use raw strings
476780 Extend strlcat and strlcpy wrappers to GNU libc
476787 Build of Valgrind 3.21.0 fails when SOLARIS_PT_SUNDWTRACE_THRP is defined
476887 WARNING: unhandled amd64-freebsd syscall: 578
477198 Add fchmodat2 syscall on linux
477628 Add mremap support for Solaris
477630 Include ucontext.h rather than sys/ucontext.h in Solaris sources
477719 vgdb incorrectly replies to qRcmd packet
478211 Redundant code for vgdb.c and Valgrind core tools
478624 Valgrind incompatibility with binutils-2.42 on x86 with new nop patterns (unhandled instruction bytes: 0x2E 0x8D 0xB4 0x26
478837 valgrind fails to read debug info for rust binaries
479041 Executables without RW sections do not trigger debuginfo reading
480052 WARNING: unhandled amd64-freebsd syscall: 580
480126 Build failure on Raspberry Pi 5 / OS 6.1.0-rpi7-rpi-v8
480405 valgrind 3.22.0 "m_debuginfo/image.c:586 (set_CEnt): Assertion '!sr_isError(sr)' failed."
480488 Add support for FreeBSD 13.3
480706 Unhandled syscall 325 (mlock2)
481127 amd64: Implement VFMADD213 for Iop_MAddF32
481131 [PATCH] x86 regtest: fix clobber lists in generated asm statements
481676 Build failure on Raspberry Pi 5 Ubuntu 23.10 with clang
481874 Add arm64 support for FreeBSD
483786 Incorrect parameter indexing in FreeBSD clock_nanosleep syscall wrapper
484002 Add suppression for invalid read in glibc's __wcpncpy_avx2() via wcsxfrm()
484426 aarch64: 0.5 gets rounded to 0
484480 False positives when using sem_trywait
484935 [patch] Valgrind reports false "Conditional jump or move depends on uninitialised value" errors for aarch64 signal handlers
485148 vfmadd213ss instruction is instrumented incorrectly (the remaining part of the register is cleared instead of kept unmodified)

485487 glibc built with -march=x86-64-v3 does not work due to ld.so strcmp
485778 Crash with --track-fds=all and --gen-suppressions=all
n-i-bz Add redirect for memccpy

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

(3.23.0.RC1: 19 Apr 2024)
(3.23.0.RC2: 24 Apr 2024)

Release 3.22.0 (31 Oct 2023)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* A new configure option --with-gdbscripts-dir lets you install
  the gdb valgrind python monitor scripts in a specific location.
  For example a distro could use it to install the scripts in a
  safe load location --with-gdbscripts-dir=%{_datadir}/gdb/auto-load
  It is also possible to configure --without-gdb-scripts-dir so no
  .debug_gdb_scripts section is added to the vgpreload library and
  no valgrind-monitor python scripts are installed at all.

* ================== PLATFORM CHANGES =================

* Support has been added for FreeBSD 14 and FreeBSD 15.
* Add support for the folllowing FreeBSD system calls:
  close_range, kqueuex, membarrier, timerfd_create,
  timerfd_settime and timerfd_gettime (all added in FreeBSD 15).

* ==================== TOOL CHANGES ===================

* Memcheck now tests and warns about the values used for
  alignment and size. These apply to various functions: memalign,
  posix_memalign and aligned_alloc in C and various overloads
  of operators new and delete in C++. The kinds of error that can
  be detected are
  - invalid alignment, for instance the alignment is usually required
    to be a power of 2
  - mismatched alignment between aligned allocation and aligned
    deallocation
  - mismatched size when sized delete is used
  - bad size for functions that have implementation defined behaviour
    when the requested size is zero

* Cachegrind:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- You can now profile part of a program's execution using the new `CACHEGRIND_START_INSTRUMENTATION` and `CACHEGRIND_STOP_INSTRUMENTATION` client requests, along with the new `--instr-at-start` option. The behaviour is the same as Callgrind's equivalent functionality.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

390871 ELF debug info reader confused with multiple .rodata* sections
417993 vbit-test fail on s390x with Iop_Add32: spurious dependency on uninit
426751 Valgrind reports "still reachable" memory using musl
(alpine running inside docker)
432801 Valgrind 3.16.1 reports a jump based on uninitialized memory somehow
related to clang and signals
433857 Add validation to C++17 aligned new/delete alignment size
433859 Add mismatched detection to C++ 17 aligned new/delete
460192 Add epoll_pwait2
461074 DWARF2 CFI reader: unhandled DW_OP_ 0x11 (consts) DW_OP_ 0x92 (bregx)
465782 s390x: Valgrind doesn't compile with Clang on s390x
466105 aligned_alloc problems, part 2
467441 Add mismatched detection to C++ 14 sized delete
469049 link failure on ppc64 (big endian) valgrind 3.20
469146 massif --ignore-fn does not ignore inlined functions
469768 Make it possible to install gdb scripts in a different location
470121 Can't run callgrind_control with valgrind 3.21.0 because of perl errors
470132 s390x: Assertion failure on VGM instruction
470520 Multiple realloc zero errors crash in MC_(eq_Error)
470713 Failure on the Yosys project: valgrind: m_libcfile.c:1802
(Bool vgPlain_realpath(const HChar *, HChar *)):
Assertion 'resolved' failed
470830 Don't print actions vgdb me ... continue for vgdb --multi mode
470978 s390x: Valgrind cannot start qemu-kvm when "sysctl vm.allocate_pgste=0"
471311 gdb --multi mode stdout redirecting to stderr
471807 Add support for lazy reading and downloading of DWARF debuginfo
472219 Syscall param ppoll(ufds.events) points to uninitialised byte(s)
472875 none/tests/s390x/dfp-1 failure
472963 Broken regular expression in configure.ac
473604 Fix bug472219.c compile failure with Clang 16
473677 make check compile failure with Clang 16 based on GCC 13.x
473745 must-be-redirected function - strlen
473870 FreeBSD 14 applications fail early at startup
473944 Handle mold linker split RW PT_LOAD segments correctly
474332 aligned_alloc under Valgrind returns nullptr when alignment is not a multiple of sizeof(void *)
475650 DRD does not work with C11 threads
475652 Missing suppression for __wcsncpy_avx2 (strncpy-avx2.S:308)?
476108 vg_replace_malloc DELETE checks size
n-i-bz Allow arguments with spaces in .valgrindrc files
n-i-bz FreeBSD fixed reading of Valgrind tools own debuginfo

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

(3.22.0.RC1: 17 Oct 2023)
(3.22.0.RC2: 26 Oct 2023)

Release 3.21.0 (28 Apr 2023)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,
PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,
MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,
X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and
AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13,
AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* When GDB is used to debug a program running under valgrind using
  the valgrind gdbserver, GDB will automatically load some
  python code provided in valgrind defining GDB front end commands
  corresponding to the valgrind monitor commands.
  These GDB front end commands accept the same format as
  the monitor commands directly sent to the Valgrind gdbserver.
  These GDB front end commands provide a better integration
  in the GDB command line interface, so as to use for example
  GDB auto-completion, command specific help, searching for
  a command or command help matching a regexp, ...
  For relevant monitor commands, GDB will evaluate arguments
  to make the use of monitor commands easier.
  For example, instead of having to print the address of a variable
  to pass it to a subsequent monitor command, the GDB front end
  command will evaluate the address argument. It is for example
  possible to do:
    (gdb) memcheck who_points_at &some_struct sizeof(some_struct)
  instead of:
    (gdb) p &some_struct
    $2 = (some_struct_type *) 0x1130a0 <some_struct>
    (gdb) p sizeof(some_struct)
    $3 = 40
    (gdb) monitor who_point_at 0x1130a0 40

* The vgdb utility now supports extended-remote protocol when
  invoked with --multi. In this mode the GDB run command is
  supported. Which means you don't need to run gdb and valgrind
  from different terminals. So for example to start your program
  in gdb and run it under valgrind you can do:
  $ gdb prog
  (gdb) set remote exec-file prog
  (gdb) set sysroot /
  (gdb) target extended-remote | vgdb --multi
  (gdb) start
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

* The behaviour of realloc with a size of zero can now
  be changed for tools that intercept malloc. Those
  tools are memcheck, helgrind, drd, massif and dhat.
  Realloc implementations generally do one of two things
    - free the memory like free() and return NULL
      (GNU libc and ptmalloc).
    - either free the memory and then allocate a
      minimum sized block or just return the
      original pointer. Return NULL if the
      allocation of the minimum sized block fails
      (jemalloc, musl, snmalloc, Solaris, macOS).
  When Valgrind is configured and built it will
  try to match the OS and libc behaviour. However
  if you are using a non-default library to replace
  malloc and family (e.g., musl on a glibc Linux or
  tcmalloc on FreeBSD) then you can use a command line
  option to change the behaviour of Valgrind:
    --realloc-zero-bytes-frees=yes|no [yes on Linux glibc, no otherwise]

* ================== PLATFORM CHANGES =================

* Make the address space limit on FreeBSD amd64 128Gbytes
  (the same as Linux and Solaris, it was 32Gbytes)

* ==================== TOOL CHANGES ===================

* Memcheck:
  - When doing a delta leak_search, it is now possible to only
    output the new loss records compared to the previous leak search.
    This is available in the memcheck monitor command 'leak_search'
    by specifying the "new" keyword or in your program by using
    the client request VALGRIND_DO_NEW_LEAK_CHECK.
    Whenever a "delta" leak search is done (i.e. when specifying
    "new" or "increased" or "changed" in the monitor command),
    the new loss records have a "new" marker.
  - Valgrind now contains python code that defines GDB memcheck
    front end monitor commands. See CORE CHANGES.
  - Performs checks for the use of realloc with a size of zero.
    This is non-portable and a source of errors. If memcheck
    detects such a usage it will generate an error
      realloc() with size 0
    followed by the usual callstacks.
    A switch has been added to allow this to be turned off:
      --show-realloc-size-zero=yes|no [yes]

* Helgrind:
  - The option ---history-backtrace-size=<number> allows to configure
    the number of entries to record in the stack traces of "old"
    accesses. Previously, this number was hardcoded to 8.
  - Valgrind now contains python code that defines GDB helgrind
    front end monitor commands. See CORE CHANGES.

* Cachegrind:

- `--cache-sim=no` is now the default. The cache simulation is old and
  unlikely to match any real modern machine. This means only the `Ir`
  event are gathered by default, but that is by far the most useful
  event.
- `cg_annotate`, `cg_diff`, and `cg_merge` have been rewritten in
  Python. As a result, they all have more flexible command line
  argument handling, e.g. supporting `--show-percs` and
  `--no-show-percs` forms as well as the existing `--show-percs=yes`
  and `--show-percs=no`.
- `cg_annotate` has some functional changes.
  - It's much faster, e.g. 3-4x on common cases.
  - It now supports diffing (with `--diff`, `--mod-filename`, and
    `--mod-funcname`) and merging (by passing multiple data files).
  - It now provides more information at the file and function level.
    There are now "File:function" and "Function:file" sections. These
    are very useful for programs that use inlining a lot.
  - Support for user-annotated files and the `-I`/`--include` option
    has been removed, because it was of little use and blocked other
    improvements.
  - The `--auto` option is renamed `--annotate`, though the old
    `--auto=yes`/`--auto=no` forms are still supported.
- `cg_diff` and `cg_merge` are now deprecated, because `cg_annotate`
  now does a better job of diffing and merging.
- The Cachegrind output file format has changed very slightly, but in
  ways nobody is likely to notice.

* Callgrind:
  - Valgrind now contains python code that defines GDB callgrind
    front end monitor commands. See CORE CHANGES.

* Massif:
  - Valgrind now contains python code that defines GDB massif
    front end monitor commands. See CORE CHANGES.

* DHAT:
  - A new kind of user request has been added which allows you to
    override the 1024 byte limit on access count histograms for blocks
    of memory. The client request is DHAT_HISTOGRAM_MEMORY.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

170510 Don't warn about ioctl of size 0 without direction hint
241072 List tools in --help output
327548 false positive while destroying mutex
382034 Testcases build fixes for musl
351857 confusing error message about valid command line option
374596 inconsistent RDTSCP support on x86_64

392331 Spurious lock not held error from inside pthread_cond_timedwait
397083 Likely false positive "uninitialised value(s)" for __wmemchr_avx2 and __wmemcmp_avx2_movbe
400793 pthread_rwlock_timedwrlock false positive
419054 Unhandled syscall getcpu on arm32
433873 openat2 syscall unimplemented on Linux
434057 Add stdio mode to valgrind's gdbserver
435441 valgrind fails to interpose malloc on musl 1.2.2 due to weak symbol name and no libc soname
436413 Warn about realloc of size zero
439685 compiler warning in callgrind/main.c
444110 priv/guest_ppc_toIR.c:36198:31: warning: duplicated 'if' condition.
444487 hginfo test detects an extra lock inside data symbol "_rtld_local"
444488 Use glibc.pthread.stack_cache_size tunable
444568 drd/tests/pth_barrier_thr_cr fails on Fedora 38
445743 "The impossible happened: mutex is locked simultaneously by two threads"
while using mutexes with priority inheritance and signals
449309 Missing loopback device ioctl(s)
459476 vgdb: allow address reuse to avoid "address already in use" errorsuse" errors
460356 s390: Sqrt32Fx4 -- cannot reduce tree
462830 WARNING: unhandled amd64-freebsd syscall: 474
463027 broken check for MPX instruction support in assembler
464103 Enhancement: add a client request to DHAT to mark memory to be histogrammed
464476 Firefox fails to start under Valgrind
464609 Valgrind memcheck should support Linux pidfd_open
464680 Show issues caused by memory policies like selinux deny_execmem
464859 Build failures with GCC-13 (drd tsan_unittest)
464969 D language demangling
465435 m_libcfile.c:66 (vgPlain_safe_fd): Assertion 'newfd >= VG_(fd_hard_limit)' failed.
466104 aligned_alloc problems, part 1
467036 Add time cost statistics for Regtest
467482 Build failure on aarch64 Alpine
467714 fdleak_* and rlimit tests fail when parent process has more than
64 descriptors opened
467839 Gdbserver: Improve compatibility of library directory name
468401 [PATCH] Add a style file for clang-format
468556 Build failure for vgdb
468606 build: remove "Valgrind relies on GCC" check/output
469097 ppc64(be) doesn't support SCV syscall instruction
n-i-bz FreeBSD rfork syscall fail with EINVAL or ENOSYS rather than VG_(unimplemented)

To see details of a given bug, visit
https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

* =================== KNOWN ISSUES ===================

* configure --enable-lto=yes is know to not work in all setups.
See bug 469049. Workaround: Build without LTO.

(3.21.0.RC1: 14 Apr 2023)
(3.21.0.RC2: 21 Apr 2023)

Release 3.20.0 (24 Oct 2022)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,
PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,
MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,
X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and
AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13,
AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* The option "--vgdb-stop-at=event1,event2,..." accepts the new value abexit.
  This indicates to invoke gdbserver when your program exits abnormally
  (i.e. with a non zero exit code).
* Fix Rust v0 name demangling.
* The Linux rseq syscall is now implemented as (silently) returning ENOSYS.
* Add FreeBSD syscall wrappers for __specialfd and __realpathat.
* Remove FreeBSD dependencies on COMPAT10, which fixes compatibility with
  HardenedBSD
* The option --enable-debuginfod=<no|yes> [default: yes] has been added on
  Linux.
* More DWARF5 support as generated by clang14.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

131186 writev reports error in (vector[...])
434764 iconv_open causes ld.so v2.28+ to use optimised strncmp
446754 Improve error codes from alloc functions under memcheck
452274 memcheck crashes with Assertion 'sci->status.what == SsIdle' failed
452779 Valgrind fails to build on FreeBSD 13.0 with llvm-devel (15.0.0)
453055 shared_timed_mutex drd test fails with "Lock shared failed" message
453602 Missing command line option to enable/disable debuginfod
452802 Handle lld 9+ split RW PT_LOAD segments correctly
454040 s390x: False-positive memcheck:cond in memmem on arch13 systems
456171 [PATCH] FreeBSD: Don't record address errors when accessing the 'kern.ps_strings' sysctl struct
n-i-bz Implement vgdb invoker on FreeBSD
458845 PowerPC: The L field for the dcbf and sync instruction should be
       3 bits in ISA 3.1.
458915 Remove register cache to fix 458915 gdbserver causes wrong syscall return
459031 Documentation on --error-exitcode incomplete
459477 XERROR messages lacks ending '\n' in vgdb
462007 Implicit int in none/tests/faultstatus.c

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

(3.20.0.RC1: 20 Oct 2022)

Release 3.19.0 (11 Apr 2022)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* Fix Rust v0 name demangling.
* The Linux rseq syscall is now implemented as (silently) returning ENOSYS.
* Add FreeBSD syscall wrappers for __specialfd and __realpathat.
* Remove FreeBSD dependencies on COMPAT10, which fixes compatibility with HardenedBSD

* ================== PLATFORM CHANGES =================

* arm64:
  - ignore the "v8.x" architecture levels, only look at actual CPU features present. Fixes mismatch detected between RDMA and atomics features preventing startup on some QEMU configurations.
  - Implement LD{,A}XP and ST{,L}XP
  - Fix incorrect code emitted for doubleword CAS.

* s390:
  - Fix sys_ipc semtimedop syscall
  - Fix VFLRX and WFLRX instructions
  - Fix EXRL instruction with negative offset

* ppc64:
  - Reimplement the vbpermq instruction support to generate less Iops and avoid overflowing internal buffers.
  - Fix checking for scv support to avoid "Facility 'SCV' unavailable (12), exception" messages in dmsg.
  - Fix setting condition code for Vector Compare quad word instructions.
  - Fix fix lxsibzx, lxsihzx and lxsihzx instructions so they only load their respective sized data.
  - Fix the prefixed stq instruction in PC relative mode.

* ==================== TOOL CHANGES ===================

* Memcheck:
  - Speed up --track-origins=yes for large (in the range of hundreds to thousands of megabytes) mmap/munmaps.
* DRD/Helgrind:
  - Several fixes for new versions of libstd++ using new posix try_lock functions

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

403802 leak_cpp_interior fails with some reachable blocks different than expected
435732 memcheck/tests/leak_cpp_interior fails with gcc11
444242 s390x: Valgrind crashes on EXRL with negative offset
444399 arm64: unhandled instruction 0xC87F2D89 (LD{,A}XP and ST{,L}XP).
== 434283
444481 gdb_server test failures on s390x
444495 dhat/tests/copy fails on s390x
444552 memcheck/tests/sem fails on s390x with glibc 2.34
444571 PPC, fix the lxsibzx and lxsihzx so they only load their respective
sized data.
444836 PPC, pstq instruction for R=1 is not storing to the correct address.
444925 fexecve syscall wrapper not properly implemented
445032 valgrind/memcheck crash with SIGSEGV when SIGVTALRM timer used and
libthr.so associated
445211 Fix out of tree builds
445300 [PATCH] Fix building tests with Musl
445011 SIGCHLD is sent when valgrind uses debuginfod-find
445354 arm64 backend: incorrect code emitted for doubleword CAS
445415 arm64 front end: alignment checks missing for atomic instructions
445504 Using C++ condition_variable results in bogus "mutex is locked simultaneously by two threads" warning
445607 Unhandled amd64-freebsd syscall: 247
445668 Inline stack frame generation is broken for Rust binaries
445916 Demangle Rust v0 symbols with .llvm suffix
446139 DRD/Helgrind with std::shared_timed_mutex::try_lock_until and try_lock_shared_until false positives
446138 DRD/Helgrind with std::timed_mutex::try_lock_until false positives
446281 Add a DRD suppression for fwrite
446103 Memcheck: `--track-origins=yes` causes extreme slowdowns for large mmap/munmap
446139 DRD/Helgrind with std::shared_timed_mutex::try_lock_until and try_lock_shared_until false
446251 TARGET_SIGNAL_THR added to enum target_signal
446823 FreeBSD - missing syscalls when using libzm4
447991 s390x: Valgrind indicates illegal instruction on wflrx
447995 Valgrind segfault on power10 due to hwcap checking code
449483 Powerpc: vcmpgtsq., vcmpgtuq,, vcmpequq. instructions not setting the
condition code correctly.
449672 ppc64 --track-origins=yes failures because of bad cmov addHRegUse
449838 sigsegv liburing the 'impossible' happened for io_uring_setup
450025 Powerc: ACC file not implemented as a logical overlay of the VSR
registers.
450437 Warn for execve syscall with argv or argv[0] being NULL
450536 Powerpc: valgrind throws 'facility scv unavailable exception'
451626 Syscall param bpf(attr->raw_tracepoint.name) points to unaddressable byte(s)
451827 [ppc64le] VEX temporary storage exhausted with several vbpermq instructions
451843 valgrind fails to start on a FreeBSD system which enforces W^X

To see details of a given bug, visit
https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed above.

(3.19.0.RC1: 02 Apr 2022)
(3.19.0.RC2: 08 Apr 2022)

3. OLDER NEWS

Release 3.18.0 (15 Oct 2021)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* The libiberty demangler has been updated, which brings support for Rust v0 name demangling. [Update: alas, due to a bug, this support isn't working in 3.18.0.]

* __libc_freeres isn't called anymore after the program receives a fatal signal. Causing some internal glibc resources to hang around, but preventing any crashes after the program has ended.

* The DWARF reader is now very much faster at startup when just --read-inline-info=yes (the default in most cases) is given.

* glibc 2.34, which moved various functions from libpthread.so into libc.so, is now supported.

* ================== PLATFORM CHANGES =================

* arm64:

  - v8.2 scalar and vector FABD, FACGE, FACGT and FADD.
  - v8.2 FP compare & conditional compare instructions.
  - Zero variants of v8.2 FP compare instructions.

* s390:

  - Support the miscellaneous-instruction-extensions facility 3 and the vector-enhancements facility 2. This enables programs compiled with "-march=arch13" or "-march=z15" to be executed under Valgrind.

* ppc64:

  - ISA 3.1 support is now complete
  - ISA 3.0 support for the darn instruction added.
  - ISA 3.0 support for the vector system call instruction scv added.
  - ISA 3.0 support for the copy, paste and cpabort instructions added.

* Support for X86/FreeBSD and AMD64/FreeBSD has been added.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

* ==================== OTHER CHANGES ====================

* Memcheck on amd64: minor fixes to remove some false positive
  undef-value errors

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

208531 [PATCH]: FreeBSD support for valgrind
368960 WARNING: unhandled amd64-linux syscall: 163 (acct)
407589 [Linux] Add support for C11 aligned_alloc() and GNU reallocarray()
423963 Error in child thread when CLONE_PIDFD is used
426148 crash with "impossible happened" when running BPF CO-RE programs
429375 PPC ISA 3.1 support is missing, part 9
431157 PPC_FEATURE2_SCV needs to be masked in AT_HWCAP2
431306 Update demangler to support Rust v0 name mangling
432387 s390x: z15 instructions support
433437 FreeBSD support, part 1
433438 FreeBSD support, part 2
433439 FreeBSD support, part 3
433469 FreeBSD support, part 4
433473 FreeBSD support, part 5
433477 FreeBSD support, part 6
433479 FreeBSD support, part 7
433504 FreeBSD support, part 8
433506 FreeBSD support, part 9
433507 FreeBSD support, part 10
433508 FreeBSD support, part 11
433510 FreeBSD support, part 12
433801 PPC ISA 3.1 support is missing, part 10 (ISA 3.1 support complete)
433863 s390x: memcheck/tests/s390x/{cds,cs,csg} failures
434296 s390x: False-positive memcheck diagnostics from vector string
       instructions
434840 PPC64 darn instruction not supported
435665 PPC ISA 3.0 copy, paste, cpabort instructions are not supported
435908 valgrind tries to fetch from deubginfod for files which already
       have debug information
438871 unhandled instruction bytes: 0xF3 0x49 0xF 0x6F 0x9C 0x24 0x60 0x2
439046 valgrind is unusably large when linked with lld
439090 Implement close_range(2)
439326 Valgrind 3.17.0 won't compile with Intel 2021 oneAPI compilers
439590 glibc-2.34 breaks suppressions against obj:*/lib*/libc-2.*so*
440670 unhandled ppc64le-linux syscall: 252 statfs64 and 253 fstatfs64
440906 Fix impossible constraint issue in P10 testcase.
441512 Remove a unneeded / unnecessary prefix check.
441534 Update the expected output for test_isa_3_1_VRT.
442061 very slow execution under Fedora 34 (readdwarf3)

```
443031  Gcc -many change requires explicit .machine directives
443033  Add support for the ISA 3.0 mcrxrx instruction
443034  Sraw, srawi, srad, sradi, mfs
443178  Powerpc, test jm-mfspr expected output needs to be updated.
443179  Need new test for the lxvx and stxvx instructions on ISA 2.07 and
        ISA 3.0 systems.
443180  The subnormal test and the ISA 3.0 test generate compiler warnings
443314  In the latest GIT version, Valgrind with "--trace-flags" crashes
        at "al" register
443605  Don't call final_tidyup (__libc_freeres) on FatalSignal

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

(3.18.0.RC1: 12 Oct 2021)
(3.18.0:     15 Oct 2021)



Release 3.17.0 (19 Mar 2021)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.17.0 fixes a number of bugs and adds some functional changes: support for
GCC 11, Clang 11, DWARF5 debuginfo, the 'debuginfod' debuginfo server, and
some new instructions for Arm64, S390 and POWER.  There are also some tool
updates.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,
PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,
MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,
X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12.  There is also preliminary
support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* DWARF version 5 support.  Valgrind can now read DWARF version 5 debuginfo as
  produced by GCC 11.

* Valgrind now supports debuginfod, an HTTP server for distributing ELF/DWARF
  debugging information. When a debuginfo file cannot be found locally,
  Valgrind is able to query debuginfod servers for the file using its
  build-id. See the user manual for more information about debuginfod support.

* ================== PLATFORM CHANGES =================

* arm64:

  - Inaccuracies resulting from double-rounding in the simulation of
    floating-point multiply-add/subtract instructions have been fixed.  These
    should now behave exactly as the hardware does.

  - Partial support for the ARM v8.2 instruction set.  v8.2 support work is
    ongoing.  Support for the half-word variants of at least the following
```

instructions has been added:
FABS <Hd>, <Hn>
FABS <Vd>.<T>, <Vn>.<T>
FNEG <Hd>, <Hn>
FNEG <Vd>.<T>, <Vn>.<T>
FSQRT <Hd>, <Hn>
FSQRT <Vd>.<T>, <Vn>.<T>
FADDP

* s390:

- Implement the new instructions/features that were added to z/Architecture with the vector-enhancements facility 1. Also cover the instructions from the vector-packed-decimal facility that are defined outside the chapter "Vector Decimal Instructions", but not the ones from that chapter itself.

  For a detailed list of newly supported instructions see the updates to `docs/internals/s390-opcodes.csv'.

  Since the miscellaneous instruction extensions facility 2 was already added in Valgrind 3.16.0, this completes the support necessary to run general programs built with `--march=z14' under Valgrind. The vector-packed-decimal facility is currently not exploited by the standard toolchain and libraries.

* ppc64:

- Various bug fixes. Fix for the sync field to limit setting just two of the two bits in the L-field. Fix the write size for the stxsibx and stxsihx instructions. Fix the modsw and modsd instructions.

- Partial support for ISA 3.1 has been added. Support for the VSX PCV mask instructions, bfloat16 GER instructions, and bfloat16 to/from float 32-bit conversion instructions are still missing.

* ==================== TOOL CHANGES ====================

* General tool changes

- All the tools and their vgpreload libraries are now installed under libexec because they cannot be executed directly and should be run through the valgrind executable. This should be an internal, not user visible, change, but might impact valgrind packagers.

- The --track-fds option now respects -q, --quiet and won't output anything if no file descriptors are leaked. It also won't report the standard stdin (0), stdout (1) or stderr (2) descriptors as being leaked with --trace-fds=yes anymore. To track whether the standard file descriptors are still open at the end of the program run use --trace-fds=all.

* DHAT:

- DHAT has been extended, with two new modes of operation. The new --mode=copy flag triggers copy profiling, which records calls to memcpy,

strcpy, and similar functions. The new --mode=ad-hoc flag triggers ad hoc
profiling, which records calls to the DHAT_AD_HOC_EVENT client request in
the new dhat/dhat.h file. This is useful for learning more about hot code
paths. See the user manual for more information about the new modes.

- Because of these changes, DHAT's file format has changed. DHAT output
files produced with earlier versions of DHAT will not work with this
version of DHAT's viewer, and DHAT output files produced with this version
of DHAT will not work with earlier versions of DHAT's viewer.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

140178 open("/proc/self/exe", ...); doesn't quite work
140939 --track-fds reports leakage of stdout/in/err and doesn't respect -q
217695 malloc/calloc/realloc/memalign failure doesn't set errno to ENOMEM
338633 gdbserver_tests/nlcontrolc.vgtest hangs on arm64
345077 linux syscall execveat support (linux 3.19)
361770 Missing F_ADD_SEALS
369029 handle linux syscalls sched_getattr and sched_setattr
384729 __libc_freeres inhibits cross-platform valgrind
388787 Support for C++17 new/delete
391853 Makefile.all.am:L247 and @SOLARIS_UNDEF_LARGESOURCE@ being empty
396656 Warnings while reading debug info
397605 ioctl FICLONE mishandled
401416 Compile failure with openmpi 4.0
408663 Suppression file for musl libc
404076 s390x: z14 vector instructions not implemented
410743 shmat() calls for 32-bit programs fail when running in 64-bit valgrind
(actually affected all x86 and nanomips regardless of host bitness)
413547 regression test does not check for Arm 64 features.
414268 Enable AArch64 feature detection and decoding for v8.x instructions
415293 Incorrect call-graph tracking due to new _dl_runtime_resolve_xsave*
422174 unhandled instruction bytes: 0x48 0xE9 (REX prefixed JMP instruction)
422261 platform selection fails for unqualified client name
422623 epoll_ctl warns for uninitialized padding on non-amd64 64bit arches
423021 PPC: Add missing ISA 3.0 documentation link and HWCAPS test.
423195 PPC ISA 3.1 support is missing, part 1
423361 Adds io_uring support on arm64/aarch64 (and all other arches)
424012 crash with readv/writev having invalid but not NULL arg2 iovec
424298 amd64: Implement RDSEED
425232 PPC ISA 3.1 support is missing, part 2
425820 Failure to recognize vpcmpeqq as a dependency breaking idiom.
426014 arm64: implement fmadd and fmsub as Iop_MAdd/Sub

426123 PPC ISA 3.1 support is missing, part 3
426144 Fix "condition variable has not been initialized" on Fedora 33.
427400 PPC ISA 3.1 support is missing, part 4
427401 PPC ISA 3.1 support is missing, part 5
427404 PPC ISA 3.1 support is missing, part 6
427870 lmw, lswi and related PowerPC insns aren't allowed on ppc64le
427787 Support new faccessat2 linux syscall (439)
427969 debuginfo section duplicates a section in the main ELF file
428035 drd: Unbreak the musl build
428648 s390_emit_load_mem panics due to 20-bit offset for vector load
428716 cppcheck detects potential leak in VEX/useful/smchash.c
428909 helgrind: need to intercept duplicate libc definitions for Fedora 33
429352 PPC ISA 3.1 support is missing, part 7
429354 PPC ISA 3.1 support is missing, part 8
429692 unhandled ppc64le-linux syscall: 147 (getsid)
429864 s390x: C++ atomic test_and_set yields false-positive memcheck
diagnostics
429952 Errors when building regtest with clang
430354 ppc stxsibx and stxsihx instructions write too much data
430429 valgrind.h doesn't compile on s390x with clang
430485 expr_is_guardable doesn't handle Iex_Qop
431556 Complete arm64 FADDP v8.2 instruction support
432102 Add support for DWARF5 as produced by GCC11
432161 Addition of arm64 v8.2 FADDP, FNEG and FSQRT
432381 drd: Process STACK_REGISTER client requests
432552 [AArch64] invalid error emitted for pre-decremented byte/hword addresses
432672 vg_regtest: test-specific environment variables not reset between tests
432809 VEX should support REX.W + POPF
432861 PPC modsw and modsd give incorrect results for 1 mod 12
432870 gdbserver_tests:nlcontrolc hangs with newest glibc2.33 x86-64
432215 Add debuginfod functionality
433323 Use pkglibexecdir as vglibdir
433500 DRD regtest faulures when libstdc++ and libgcc debuginfo are installed
433629 valgrind/README has type "abd" instead of "and"
433641 Rust std::sys::unix::fs::try_statx Syscall param fstatat(file_name)
433898 arm64: Handle sp, lr, fp as DwReg in CfiExpr
434193 GCC 9+ inlined strcmp causes "Conditional jump or move [..] value" report
n-i-bz helgrind: If hg_cli__realloc fails, return NULL.
n-i-bz arm64 front end: avoid Memcheck false positives relating to CPUID

(3.17.0.RC1: 13 Mar 2021)
(3.17.0.RC2: 17 Mar 2021)
(3.17.0:   19 Mar 2021)

Release 3.16.1 (22 June 2020)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.16.1 fixes two critical bugs discovered after 3.16.0 was frozen. It also
fixes character encoding problems in the documentation HTML.

422677 PPC sync instruction L field should only be 2 bits in ISA 3.0
422715 32-bit x86: vex: the `impossible' happened: expr_is_guardable: unhandled expr
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(3.16.1, 22 June 2020, 36d6727e1d768333a536f274491e5879cab2c2f7)

Release 3.16.0 (27 May 2020)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.16.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

* ==================== CORE CHANGES ===================

* It is now possible to dynamically change the value of many command line options while your program (or its children) are running under Valgrind.

  To see the list of dynamically changeable options, run
  "valgrind --help-dyn-options".

  You can change the options from the shell by using vgdb to launch the monitor command "v.clo <clo option>...".
  The same monitor command can be used from a gdb connected to the valgrind gdbserver.
  Your program can also change the dynamically changeable options using the client request VALGRIND_CLO_CHANGE(option).

* ================== PLATFORM CHANGES =================

* MIPS: preliminary support for nanoMIPS instruction set has been added.

* ==================== TOOL CHANGES ====================

* DHAT:

  - The implicit memcpy done by each call to realloc now counts towards the read and write counts of resized heap blocks, making those counts higher and more accurate.

* Cachegrind:

  - cg_annotate's --auto and --show-percs options now default to 'yes', because they are usually wanted.

* Callgrind:

  - callgrind_annotate's --auto and --show-percs options now default to 'yes', because they are usually wanted.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- The command option --collect-systime has been enhanced to specify
  the unit used to record the elapsed time spent during system calls.
  The command option now accepts the values no|yes|msec|usec|nsec,
  where yes is a synonym of msec. When giving the value nsec, the
  system cpu time of system calls is also recorded.

* Memcheck:

- Several memcheck options are now dynamically changeable.
  Use valgrind --help-dyn-options to list them.

- The release 3.15 introduced a backward incompatible change for
  some suppression entries related to preadv and pwritev syscalls.
  When reading a suppression entry using the unsupported 3.14 format,
  valgrind will now produce a warning to say the suppression entry will not
  work, and suggest the needed change.

- Significantly fewer false positive errors on optimised code generated by
  Clang and GCC. In particular, Memcheck now deals better with the
  situation where the compiler will transform C-level "A && B" into "B && A"
  under certain circumstances (in which the transformation is valid).
  Handling of integer equality/non-equality checks on partially defined
  values is also improved on some architectures.

* exp-sgcheck:

- The exprimental Stack and Global Array Checking tool has been removed.
  It only ever worked on x86 and amd64, and even on those it had a
  high false positive rate and was slow. An alternative for detecting
  stack and global array overruns is using the AddressSanitizer (ASAN)
  facility of the GCC and Clang compilers, which require you to rebuild
  your code with -fsanitize=address.

* ==================== OTHER CHANGES ====================

* New and modified GDB server monitor features:

- Option -T tells vgdb to output a timestamp in the vgdb information messages.

- The gdbserver monitor commands that require an address and an optional
  length argument now accepts the alternate 'C like' syntax "address[length]".
  For example, the memcheck command "monitor who_points_at 0x12345678 120"
  can now also be given as "monitor who_points_at 0x12345678[120]".

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

343099 Linux setns syscall wrapper missing, unhandled syscall: 308
 == 368923 WARNING: unhandled arm64-linux syscall: 268 (setns)
 == 369031 WARNING: unhandled amd64-linux syscall: 308 (setns)
385386 Assertion failed "szB >= CACHE_ENTRY_SIZE" at m_debuginfo/image.c:517
400162 Patch: Guard against __GLIBC_PREREQ for musl libc
400593 In Coregrind, use statx for some internal syscalls if [f]stat[64] fail
400872 Add nanoMIPS support to Valgrind
403212 drd/tests/trylock hangs on FreeBSD
404406 s390x: z14 miscellaneous instructions not implemented
405201 Incorrect size of struct vki_siginfo on 64-bit Linux architectures
406561 mcinfcallWSRU gdbserver_test fails on ppc64
406824 Unsupported baseline
407218 Add support for the copy_file_range syscall
407307 Intercept stpcpy also in ld.so for arm64
407376 Update Xen support to 4.12 (4.13, actually) and add more coverage
 == 390553
407764 drd cond_post_wait gets wrong (?) condition on s390x z13 system
408009 Expose rdrand and f16c even on avx if host cpu supports them
408091 Missing pkey syscalls
408414 Add support for missing for preadv2 and pwritev2 syscalls
409141 Valgrind hangs when SIGKILLed
409206 Support for Linux PPS and PTP ioctls
409367 exit_group() after signal to thread waiting in futex() causes hangs
409429 amd64: recognize 'cmpeq' variants as a dependency breaking idiom
409780 References to non-existent configure.in
410556 Add support for BLKIO{MIN,OPT} and BLKALIGNOFF ioctls
410599 Non-deterministic behaviour of pth_self_kill_15_other test
410757 discrepancy for preadv2/pwritev2 syscalls across different versions
411134 Allow the user to change a set of command line options during execution
411451 amd64->IR of bt/btc/bts/btr with immediate clears zero flag
412344 Problem setting mips flags with specific paths
412408 unhandled arm-linux syscall: 124 - adjtime - on arm-linux
413119 Ioctl wrapper for DRM_IOCTL_I915_GEM_MMAP
413330 avx-1 test fails on AMD EPYC 7401P 24-Core Processor
413603 callgrind_annotate/cg_annotate truncate function names at '#'
414565 Specific use case bug found in SysRes VG_(do_sys_sigprocmask)
415136 ARMv8.1 Compare-and-Swap instructions are not supported
415757 vex x86->IR: 0x66 0xF 0xCE 0x4F (bswapw)
416239 valgrind crashes when handling clock_adjtime
416285 Use prlimit64 in VG_(getrlimit) and VG_(setrlimit)
416286 DRD reports "conflicting load" error on std::mutex::lock()
416301 s390x: "compare and signal" not supported
416387 finit_module and bpf syscalls are unhandled on arm64
416464 Fix false reports for uninitialized memory for PR_CAPBSET_READ/DROP
416667 gcc10 ppc64le impossible constraint in 'asm' in test_isa.
416753 new 32bit time syscalls for 2038+
417075 pwritev(vector[...]) suppression ignored
 417075 is not fixed, but incompatible supp entries are detected
 and a warning is produced for these.
417187 [MIPS] Conditional branch problem since 'grail' changes
417238 Test memcheck/tests/vbit-test fails on mips64 BE

417266 Make memcheck/tests/linux/sigqueue usable with musl
417281 s390x: /bin/true segfaults with "grail" enabled
417427 commit to fix vki_siginfo_t definition created numerous regression
        errors on ppc64
417452 s390_insn_store_emit: dst->tag for HRcVec128
417578 Add suppressions for glibc DTV leaks
417906 clone with CLONE_VFORK and no CLONE_VM fails
418004 Grail code additions break ppc64.
418435 s390x: spurious "Conditional jump or move depends on uninitialised [..]"
418997 s390x: Support Iex_ITE for float and vector types
419503 s390x: Avoid modifying registers returned from isel functions
421321 gcc10 arm64 build needs __getauxval for linking with libgcc
421570 std_mutex fails on Arm v8.1 h/w
434035 vgdb might crash if valgrind is killed
n-i-bz Fix minor one time leaks in dhat.
n-i-bz Add --run-cxx-freeres=no in outer args to avoid inner crashes.
n-i-bz Add support for the Linux io_uring system calls
n-i-bz sys_statx: don't complain if both |filename| and |buf| are NULL.
n-i-bz Fix non-glibc build of test suite with s390x_features
n-i-bz MinGW, include/valgrind.h: Fix detection of 64-bit mode
423195 PPC ISA 3.1 support is missing, part 1

(3.16.0.RC1: 18 May 2020, git 6052ee66a0cf5234e8e2a2b49a8760226bc13b92)
(3.16.0.RC2: 19 May 2020, git 940ec1ca69a09f7fdae3e800b7359f85c13c4b37)
(3.16.0:   27 May 2020, git bf5e647edb9e96cbd5c57cc944984402eeee296d)

Release 3.15.0 (12 April 2019)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.15.0 is a feature release with many improvements and the usual collection of
bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,
PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,
MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,
X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12. There is also preliminary
support for X86/macOS 10.13 and AMD64/macOS 10.13.

* ==================== CORE CHANGES ===================

* The XTree Massif output format now makes use of the information obtained
  when specifying --read-inline-info=yes.

* amd64 (x86_64): the RDRAND and F16C insn set extensions are now supported.

* ==================== TOOL CHANGES ====================

* DHAT:

  - DHAT been thoroughly overhauled, improved, and given a GUI. As a result,
    it has been promoted from an experimental tool to a regular tool. Run it
    with --tool=dhat instead of --tool=exp-dhat.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  - DHAT now prints only minimal data when the program ends, instead writing
    the bulk of the profiling data to a file. As a result, the --show-top-n
    and --sort-by options have been removed.

  - Profile results can be viewed with the new viewer, dh_view.html. When
    a run ends, a short message is printed, explaining how to view the result.

  - See the documentation for more details.

* Cachegrind:

  - cg_annotate has a new option, --show-percs, which prints percentages next
    to all event counts.

* Callgrind:

  - callgrind_annotate has a new option, --show-percs, which prints percentages
    next to all event counts.

  - callgrind_annotate now inserts commas in call counts, and
    sort the caller/callee lists in the call tree.

* Massif:

  - The default value for --read-inline-info is now "yes" on
    Linux/Android/Solaris. It is still "no" on other OS.

* Memcheck:

  - The option --xtree-leak=yes (to output leak result in xtree format)
    automatically activates the option --show-leak-kinds=all, as xtree
    visualisation tools such as kcachegrind can in any case select what kind
    of leak to visualise.

  - There has been further work to avoid false positives. In particular,
    integer equality on partially defined inputs (C == and !=) is now handled
    better.

* ==================== OTHER CHANGES ====================

* The new option --show-error-list=no|yes displays, at the end of the run, the
  list of detected errors and the used suppressions. Prior to this change,
  showing this information could only be done by specifying "-v -v", but that
  also produced a lot of other possibly-non-useful messages. The option -s is
  equivalent to --show-error-list=yes.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that

are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

385411 s390x: z13 vector floating-point instructions not implemented
397187 z13 vector register support for vgdb gdbserver
398183 Vex errors with _mm256_shuffle_epi8/vpshufb
398870 Please add support for instruction vcvtps2ph
399287 amd64 front end: Illegal Instruction vcmptrueps
399301 Use inlined frames in Massif XTree output.
399322 Improve callgrind_annotate output
399444 VEX/priv/guest_s390_toIR.c:17407: (style) Mismatching assignment [..]
400164 helgrind test encounters mips x-compiler warnings and assembler error
400490 s390x: VRs allocated as if separate from FPRs
400491 s390x: Operand of LOCH treated as unsigned integer
400975 Compile error: error: '-mips64r2' conflicts with the other architecture
        options, which specify a mips64 processor
401112 LLVM 5.0 generates comparison against partially initialized data
401277 More bugs in z13 support
401454 Add a --show-percs option to cg_annotate and callgrind_annotate.
401578 drd: crashes sometimes on fork()
401627 memcheck errors with glibc avx2 optimized wcsncmp
401822 none/tests/ppc64/jm-vmx fails and produces assembler warnings
401827 none/tests/ppc64/test_isa_2_06_part3 failure on ppc64le (xvrsqrtesp)
401828 none/tests/ppc64/test_isa_2_06_part1 failure on ppc64le (fcfids and
        fcfidus)
402006 mark helper regs defined in final_tidyup before freeres_wrapper call
402048 WARNING: unhandled ppc64[be|le]-linux syscall: 26 (ptrace)
402123 invalid assembler opcodes for mips32r2
402134 assertion fail in mc_translate.c (noteTmpUsesIn) Iex_VECRET on arm64
402327 Warning: DWARF2 CFI reader: unhandled DW_OP_ opcode 0x13 (DW_OP_drop)
402341 drd/tests/tsan_thread_wrappers_pthread.h:369: suspicious code ?
402351 mips64 libvexmultiarch_test fails on s390x
402369 Overhaul DHAT
402395 coregrind/vgdb-invoker-solaris.c: 2 * poor error checking
402480 Do not use %rsp in clobber list
402481 vbit-test fails on x86 for Iop_CmpEQ64 iselInt64Expr Sar64
402515 Implement new option --show-error-list=no|yes / -s
402519 POWER 3.0 addex instruction incorrectly implemented
402781 Redo the cache used to process indirect branch targets
403123 vex amd64->IR:0xF3 0x48 0xF 0xAE 0xD3 (wrfsbase)
403552 s390x: wrong facility bit checked for vector facility
404054 memcheck powerpc subfe x, x, x initializes x to 0 or -1 based on CA
404638 Add VG_(replaceIndexXA)
404843 s390x: backtrace sometimes ends prematurely
404888 autotools cleanup series
405079 unhandled ppc64le-linux syscall: 131 (quotactl)
405182 Valgrind fails to build with Clang
405205 filter_libc: remove the line holding the futex syscall error entirely
405356 PPC64, xvcvsxdsp, xvcvuxdsp are supposed to write the 32-bit result to
        the upper and lower 32-bits of the 64-bit result
405362 PPC64, vmsummbm instruction doesn't handle overflow case correctly

405363 PPC64, xvcvdpsxws, xvcvdpuxws, do not handle NaN arguments correctly.
405365 PPC64, function _get_maxmin_fp_NaN() doesn't handle QNaN, SNaN case
        correctly.
405403 s390x disassembler cannot be used on x86
405430 Use gcc -Wimplicit-fallthrough=2 by default if available
405458 MIPS mkFormVEC arguments swapped?
405716 drd: Fix an integer overflow in the stack margin calculation
405722 Support arm64 core dump
405733 PPC64, xvcvdpsp should write 32-bit result to upper and lower 32-bits
        of the 64-bit destination field.
405734 PPC64, vrlwnm, vrlwmi, vrldrm, vrldmi do not work properly when me < mb
405782 "VEX temporary storage exhausted" when attempting to debug slic3r-pe
406198 none/tests/ppc64/test_isa_3_0_other test sporadically including CA
        bit in output.
406256 PPC64, vector floating point instructions don't handle subnormal
        according to VSCR[NJ] bit setting.
406352 cachegrind/callgrind fails ann tests because of missing a.c
406354 dhat is broken on x86 (32bit)
406355 mcsignopass, mcsigpass, mcbreak fail due to difference in gdb output
406357 gdbserver_tests fails because of gdb output change
406360 memcheck/tests/libstdc++.supp needs more supression variants
406422 none/tests/amd64-linux/map_32bits.vgtest fails too easily
406465 arm64 insn selector fails on "t0 = <expr>" where <expr> has type Ity_F16
407340 PPC64, does not support the vlogefp, vexptefp instructions.
n-i-bz add syswrap for PTRACE_GET|SET_THREAD_AREA on amd64.
n-i-bz Fix callgrind_annotate non deterministic order for equal total
n-i-bz callgrind_annotate --threshold=100 does not print all functions.
n-i-bz callgrind_annotate Use of uninitialized value in numeric gt (>)
n-i-bz amd64 (x86_64): RDRAND and F16C insn set extensions are supported

(3.15.0.RC1: 8 April 2019, git ce94d674de5b99df173aad4c3ee48fc2a92e5d9c)
(3.15.0.RC2: 11 April 2019, git 0c8be9bbede189ec580ec270521811766429595f)
(3.15.0:   14 April 2019, git 270037da8b508954f0f7d703a0bebf5364eec548)

Release 3.14.0 (9 October 2018)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.14.0 is a feature release with many improvements and the usual collection of
bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,
PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,
MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,
X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12. There is also preliminary
support for X86/macOS 10.13, AMD64/macOS 10.13.

* ==================== CORE CHANGES ===================

* The new option --keep-debuginfo=no|yes (default no) can be used to retain
  debug info for unloaded code. This allows saved stack traces (e.g. for
  memory leaks) to include file/line info for code that has been dlclose'd (or
  similar). See the user manual for more information and known limitations.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

* Ability to specify suppressions based on source file name and line number.

* Majorly overhauled register allocator. No end-user changes, but the JIT
  generates code a bit more quickly now.

* ================== PLATFORM CHANGES =================

* Preliminary support for macOS 10.13 has been added.

* mips: support for MIPS32/MIPS64 Revision 6 has been added.

* mips: support for MIPS SIMD architecture (MSA) has been added.

* mips: support for MIPS N32 ABI has been added.

* s390: partial support for vector instructions (integer and string) has been
  added.

* ==================== TOOL CHANGES ====================

* Helgrind: Addition of a flag
  --delta-stacktrace=no|yes [yes on linux amd64/x86]
  which specifies how full history stack traces should be computed.
  Setting this to =yes can speed up Helgrind by 25% when using
  --history-level=full.

* Memcheck: reduced false positive rate for optimised code created by Clang 6
  / LLVM 6 on x86, amd64 and arm64. In particular, Memcheck analyses code
  blocks more carefully to determine where it can avoid expensive definedness
  checks without loss of precision. This is controlled by the flag
  --expensive-definedness-checks=no|auto|yes [auto].

* ==================== OTHER CHANGES ====================

* Valgrind is now buildable with link-time optimisation (LTO). A new
  configure option --enable-lto=yes allows building Valgrind with LTO. If the
  toolchain supports it, this produces a smaller/faster Valgrind (up to 10%).
  Note that if you are doing Valgrind development, --enable-lto=yes massively
  slows down the build process.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

79362   Debug info is lost for .so files when they are dlclose'd
208052 strlcpy error when n = 0
255603 exp-sgcheck Assertion '!already_present' failed
338252 building valgrind with -flto (link time optimisation) fails
345763 MIPS N32 ABI support
368913 WARNING: unhandled arm64-linux syscall: 117 (ptrace)
        == 388664 unhandled arm64-linux syscall: 117 (ptrace)
372347 Replacement problem of the additional c++14/c++17 new/delete operators
373069 memcheck/tests/leak_cpp_interior fails with GCC 5.1+
376257 helgrind history full speed up using a cached stack
379373 Fix syscall param msg->desc.port.name points to uninitialised byte(s)
        on macOS 10.12
379748 Fix missing pselect syscall (OS X 10.11)
379754 Fix missing syscall ulock_wait (OS X 10.12)
380397 s390x: __GI_strcspn() replacemenet needed
381162 possible array overrun in VEX register allocator
381272 ppc64 doesn't compile test_isa_2_06_partx.c without VSX support
381274 powerpc too chatty even with --sigill-diagnostics=no
381289 epoll_pwait can have a NULL sigmask
381553 VEX register allocator v3
381556 arm64: Handle feature registers access on 4.11 Linux kernel or later
381769 Use ucontext_t instead of struct ucontext
381805 arm32 needs ld.so index hardwire for new glibc security fixes
382256 gz compiler flag test doesn't work for gold
382407 vg_perf needs "--terse" command line option
382515 "Assertion 'di->have_dinfo' failed." on wine's dlls/mscoree/tests/[..]
382563 MIPS MSA ASE support
382998 xml-socket doesn't work
383275 massif: m_xarray.c:162 (ensureSpaceXA): Assertion '!xa->arr' failed
383723 Fix missing kevent_qos syscall (macOS 10.11)
        == 385604 illegal hardware instruction (OpenCV cv::namedWindow)
384096 Mention AddrCheck at Memcheck's command line option [..]
384230 vex x86->IR: 0x67 0xE8 0xAB 0x68
        == 384156 vex x86->IR: 0x67 0xE8 0x6B 0x6A
        == 386115 vex x86->IR: 0x67 0xE8 0xD3 0x8B any program
        == 388407 vex x86->IR: 0x67 0xE8 0xAB 0x29
        == 394903 vex x86->IR: 0x67 0xE8 0x1B 0xDA
384337 performance improvements to VEX register allocator v2 and v3
384526 reduce number of spill insns generated by VEX register allocator v3
384584 Callee saved regs listed first for AMD64, X86, and PPC architectures
384631 Sanitise client args as printed with -v
384633 Add a simple progress-reporting facility
384987 VEX regalloc: allocate caller-save registers for short lived vregs
385055 PPC VEX temporary storage exhausted
385182 PPC64 is missing support for the DSCR
385183 PPC64, Add support for xscmpeqdp, xscmpgtdp, xscmpgedp, xsmincdp
385207 PPC64, generate_store_FPRF() generates too many Iops
385208 PPC64, xxperm instruction exhausts temporary memory
385210 PPC64, vpermr instruction could exhaust temporary memory
385279 unhandled syscall: mach:43 (mach_generate_activity_id)
        == 395136 valgrind: m_syswrap/syswrap-main.c:438 (Bool eq_Syscall[..]
        == 387045 Valgrind crashing on High Sierra when testing any newly [..]
385334 PPC64, fix vpermr, xxperm, xxpermr mask value.
385408 s390x: z13 vector "support" instructions not implemented

385409 s390x: z13 vector integer instructions not implemented
385410 s390x: z13 vector string instructions not implemented
385412 s390x: new non-vector z13 instructions not implemented
385868 glibc ld.so _dl_runtime_resolve_avx_slow conditional jump warning.
385912 none/tests/rlimit_nofile fails on newer glibc/kernel.
385939 Optionally exit on the first error
386318 valgrind.org/info/tools.html is missing SGCheck
386425 running valgrind + wine on armv7l gives illegal opcode
386397 PPC64, valgrind truncates powerpc timebase to 32-bits.
387410 MIPSr6 support
387664 Memcheck: make expensive-definedness-checks be the default
387712 s390x cgijnl reports Conditional jump depends on uninitialised value
387766 asm shifts cause false positive "Conditional jump or move depends
    on uninitialised value"
387773 .gnu_debugaltlink paths resolve relative to .debug file, not symlink
388174 valgrind with Wine quits with "Assertion 'cfsi_fits' failed"
388786 Support bpf syscall in amd64 Linux
388862 Add replacements for wmemchr and wcsnlen on Linux
389065 valgrind meets gcc flag -Wlogical-op
389373 exp-sgcheck the 'impossible' happened as Ist_LoadG is not instrumented
390471 suppression by specification of source-file line number
390723 make xtree dump files world wide readable, similar to log files
391164 constraint bug in tests/ppc64/test_isa_2_07_part1.c for mtfprwa
391861 Massif Assertion 'n_ips >= 1 && n_ips <= VG_(clo_backtrace_size)'
392118 unhandled amd64-linux syscall: 332 (statx)
392449 callgrind not clearing the number of calls properly
393017 Add missing support for xsmaxcdp instruction, bug fixes for xsmincdp,
    lxssp, stxssp and stxvl instructions.
393023 callgrind_control risks using the wrong vgdb
393062 build-id ELF phdrs read causes "debuginfo reader: ensure_valid failed"
393099 posix_memalign() invalid write if alignment == 0
393146 failing assert "is_DebugInfo_active(di)"
395709 PPC64 is missing support for the xvnegsp instruction
395682 Accept read-only PT_LOAD segments and .rodata by ld -z separate-code
    == 384727
396475 valgrind OS-X build: config.h not found (out-of-tree macOS builds)
395991 arm-linux: wine's unit tests enter a signal delivery loop [..]
396839 s390x: Trap instructions not implemented
396887 arch_prctl should return EINVAL on unknown option
    == 397286 crash before launching binary (Unsupported arch_prctl option)
    == 397393 valgrind: the 'impossible' happened: (Archlinux)
    == 397521 valgrind: the 'impossible' happened: Unsupported [..]
396906 compile tests failure on mips32-linux: broken inline asm in tests on
    mips32-linux
397012 glibc ld.so uses arch_prctl on i386
397089 amd64: Incorrect decoding of three-register vmovss/vmovsd opcode 11h
397354 utimensat should ignore timespec tv_sec if tv_nsec is UTIME_NOW/OMIT
397424 glibc 2.27 and gdb_server tests
398028 Assertion `cfsi_fits` failing in simple C program
398066 s390x: cgijl dep1, 0 reports false unitialised values warning

n-i-bz Fix missing workq_ops operations (macOS)
n-i-bz fix bug in strspn replacement
n-i-bz Add support for the Linux BLKFLSBUF ioctl

n-i-bz Add support for the Linux BLKREPORTZONE and BLKRESETZONE ioctls
n-i-bz Fix possible stack trashing by semctl syscall wrapping
n-i-bz Add support for the Linux membarrier() system call
n-i-bz x86 front end: recognise and handle UD2 correctly
n-i-bz Signal delivery for x86-linux: ensure that the stack pointer is
        correctly aligned before entering the handler.

(3.14.0.RC1: 30 September 2018, git c2aeea2d28acb0639bcc8cc1e4ab115067db1eae)
(3.14.0.RC2: 3 October 2018, git 3e214c4858a6fdd5697e767543a0c19e30505582)
(3.14.0:   9 October 2018, git 353a3587bb0e2757411f9138f5e936728ed6cc4f)

Release 3.13.0 (15 June 2017)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.13.0 is a feature release with many improvements and the usual collection of
bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,
PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,
MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,
X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12.

* ==================== CORE CHANGES ===================

* The translation cache size has been increased to keep up with the demands of
  large applications. The maximum number of sectors has increased from 24 to
  48. The default number of sectors has increased from 16 to 32 on all
  targets except Android, where the increase is from 6 to 12.

* The amount of memory that Valgrind can use has been increased from 64GB to
  128GB. In particular this means your application can allocate up to about
  60GB when running on Memcheck.

* Valgrind's default load address has been changed from 0x3800'0000 to
  0x5800'0000, so as to make it possible to load larger executables. This
  should make it possible to load executables of size at least 1200MB.

* A massive spaceleak caused by reading compressed debuginfo files has been
  fixed. Valgrind should now be entirely usable with gcc-7.0 "-gz" created
  debuginfo.

* The C++ demangler has been updated.

* Support for demangling Rust symbols has been added.

* A new representation of stack traces, the "XTree", has been added. An XTree
  is a tree of stacktraces with data associated with the stacktraces. This is
  used by various tools (Memcheck, Helgrind, Massif) to report on the heap
  consumption of your program. Reporting is controlled by the new options
  --xtree-memory=none|allocs|full and --xtree-memory-file=<file>.

  A report can also be produced on demand using the gdbserver monitor command
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

'xtmemory [<filename>]>'. The XTree can be output in 2 formats: 'callgrind format' and 'massif format. The existing visualisers for these formats (e.g. callgrind_annotate, KCachegrind, ms_print) can be used to visualise and analyse these reports.

Memcheck can also produce XTree leak reports using the Callgrind file format. For more details, see the user manual.

* ================== PLATFORM CHANGES =================

* ppc64: support for ISA 3.0B and various fixes for existing 3.0 support

* amd64: fixes for JIT failure problems on long AVX2 code blocks

* amd64 and x86: support for CET prefixes has been added

* arm32: a few missing ARMv8 instructions have been implemented

* arm64, mips64, mips32: an alternative implementation of Load-Linked and Store-Conditional instructions has been added. This is to deal with processor implementations that implement the LL/SC specifications strictly and as a result cause Valgrind to hang in certain situations. The alternative implementation is automatically enabled at startup, as required. You can use the option --sim-hints=fallback-llsc to force-enable it if you want.

* Support for OSX 10.12 has been improved.

* On Linux, clone handling has been improved to honour CLONE_VFORK that involves a child stack. Note however that CLONE_VFORK | CLONE_VM is handled like CLONE_VFORK (by removing CLONE_VM), so applications that depend on CLONE_VM exact semantics will (still) not work.

* The TileGX/Linux port has been removed because it appears to be both unused and unsupported.

* ==================== TOOL CHANGES ====================

* Memcheck:

  - Memcheck should give fewer false positives when running optimised Clang/LLVM generated code.

  - Support for --xtree-memory and 'xtmemory [<filename>]>'.

  - New command line options --xtree-leak=no|yes and --xtree-leak-file=<file> to produce the end of execution leak report in a xtree callgrind format file.

  - New option 'xtleak' in the memcheck leak_check monitor command, to produce the leak report in an xtree file.

* Massif:

- Support for --xtree-memory and 'xtmemory [<filename>]>'.

- For some workloads (typically, for big applications), Massif memory
  consumption and CPU consumption has decreased significantly.

* Helgrind:

- Support for --xtree-memory and 'xtmemory [<filename>]>'.

- addition of client request VALGRIND_HG_GNAT_DEPENDENT_MASTER_JOIN, useful
  for Ada gnat compiled applications.

* ==================== OTHER CHANGES ====================

* For Valgrind developers: in an outer/inner setup, the outer Valgrind will
  append the inner guest stacktrace to the inner host stacktrace. This helps
  to investigate the errors reported by the outer, when they are caused by the
  inner guest program (such as an inner regtest). See README_DEVELOPERS for
  more info.

* To allow fast detection of callgrind files by desktop environments and file
  managers, the format was extended to have an optional first line that
  uniquely identifies the format ("# callgrind format"). Callgrind creates
  this line now, as does the new xtree functionality.

* File name template arguments (such as --log-file, --xtree-memory-file, ...)
  have a new %n format letter that is replaced by a sequence number.

* "--version -v" now shows the SVN revision numbers from which Valgrind was
  built.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

162848 --log-file output isn't split when a program forks
340777 Illegal instruction on mips (ar71xx)
341481 MIPS64: Iop_CmpNE32 triggers false warning on MIPS64 platforms
342040 Valgrind mishandles clone with CLONE_VFORK | CLONE_VM that clones
       to a different stack.
344139 x86 stack-seg overrides, needed by the Wine people
344524 store conditional of guest applications always fail - observed on
       Octeon3(MIPS)
348616 Wine/valgrind: noted but unhandled ioctl 0x5390 [..] (DVD_READ_STRUCT)
352395 Please provide SVN revision info in --version -v

352767 Wine/valgrind: noted but unhandled ioctl 0x5307 [..] (CDROMSTOP)
356374 Assertion 'DRD_(g_threadinfo)[tid].pt_threadid !=
INVALID_POSIX_THREADID' failed
358213 helgrind/drd bar_bad testcase hangs or crashes with new glibc pthread
barrier implementation
358697 valgrind.h: Some code remains even when defining NVALGRIND
359202 Add musl libc configure/compile
360415 amd64 instructions ADCX and ADOX are not implemented in VEX
== 372828 (vex amd64->IR: 0x66 0xF 0x3A 0x62 0x4A 0x10)
360429 unhandled ioctl 0x530d with no size/direction hints (CDROMREADMODE1)
362223 assertion failed when .valgrindrc is a directory instead of a file
367543 bt/btc/btr/bts x86/x86_64 instructions are poorly-handled wrt flags
367942 Segfault vgPlain_do_sys_sigaction (m_signals.c:1138)
368507 can't malloc chunks larger than about 34GB
368529 Android arm target link error, missing atexit and pthread_atfork
368863 WARNING: unhandled arm64-linux syscall: 100 (get_robust_list)
368865 WARNING: unhandled arm64-linux syscall: 272 (kcmp)
368868 disInstr(arm64): unhandled instruction 0xD53BE000 = cntfrq_el0 (ARMv8)
368917 WARNING: unhandled arm64-linux syscall: 218 (request_key)
368918 WARNING: unhandled arm64-linux syscall: 127 (sched_rr_get_interval)
368922 WARNING: unhandled arm64-linux syscall: 161 (sethostname)
368924 WARNING: unhandled arm64-linux syscall: 84 (sync_file_range)
368925 WARNING: unhandled arm64-linux syscall: 130 (tkill)
368926 WARNING: unhandled arm64-linux syscall: 97 (unshare)
369459 valgrind on arm64 violates the ARMv8 spec (ldxr/stxr)
370028 Reduce the number of compiler warnings on MIPS platforms
370635 arm64 missing syscall getcpu
371225 Fix order of timer_{gettime,getoverrun,settime} syscalls on arm64
371227 Clean AArch64 syscall table
371412 Rename wrap_sys_shmat to sys_shmat like other wrappers
371471 Valgrind complains about non legit memory leaks on placement new (C++)
371491 handleAddrOverrides() is [incorrect] when ASO prefix is used
371503 disInstr(arm64): unhandled instruction 0xF89F0000
371869 support '%' in symbol Z-encoding
371916 execution tree xtree concept
372120 c++ demangler demangles symbols which are not c++
372185 Support of valgrind on ARMv8 with 32 bit executable
372188 vex amd64->IR: 0x66 0xF 0x3A 0x62 0x4A 0x10 0x10 0x48 (PCMPxSTRx $0x10)
372195 Power PC, xxsel instruction is not always recognized.
372504 Hanging on exit_group
372600 process loops forever when fatal signals are arriving quickly
372794 LibVEX (arm32 front end): 'Assertion szBlg2 <= 3' failed
373046 Stacks registered by core are never deregistered
373069 memcheck/tests/leak_cpp_interior fails with GCC 5.1+
373086 Implement additional Xen hypercalls
373192 Calling posix_spawn in glibc 2.24 completely broken
373488 Support for fanotify API on ARM64 architecture
== 368864 WARNING: unhandled arm64-linux syscall: 262 (fanotify_init)
373555 Rename BBPTR to GSPTR as it denotes guest state pointer only
373938 const IRExpr arguments for matchIRExpr()
374719 some spelling fixes
374963 increase valgrind's load address to prevent mmap failure
375514 valgrind_get_tls_addr() does not work in case of static TLS
375772 +1 error in get_elf_symbol_info() when computing value of 'hi' address

for ML_(find_rx_mapping)()
375806 Test helgrind/tests/tc22_exit_w_lock fails with glibc 2.24
375839 Temporary storage exhausted, with long sequence of vfmadd231ps insns
== 377159 "vex: the `impossible' happened" still present
== 375150 Assertion 'tres.status == VexTransOK' failed
== 378068 valgrind crashes on AVX2 function in FFmpeg
376142 Segfaults on MIPS Cavium Octeon boards
376279 disInstr(arm64): unhandled instruction 0xD50320FF
376455 Solaris: unhandled syscall lgrpsys(180)
376518 Solaris: unhandled fast trap getlgrp(6)
376611 ppc64 and arm64 don't know about prlimit64 syscall
376729 PPC64, remove R2 from the clobber list
== 371668
376956 syswrap of SNDDRV and DRM_IOCTL_VERSION causing some addresses
to be wrongly marked as addressable
377066 Some Valgrind unit tests fail to compile on Ubuntu 16.10 with
PIE enabled by default
377376 memcheck/tests/linux/getregset fails with glibc2.24
377427 PPC64, lxv instruction failing on odd destination register
377478 PPC64: ISA 3.0 setup fixes
377698 Missing memory check for futex() uaddr arg for FUTEX_WAKE
and FUTEX_WAKE_BITSET, check only 4 args for FUTEX_WAKE_BITSET,
and 2 args for FUTEX_TRYLOCK_PI
377717 Fix massive space leak when reading compressed debuginfo sections
377891 Update Xen 4.6 domctl wrappers
377930 fcntl syscall wrapper is missing flock structure check
378524 libvexmultiarch_test regression on s390x and ppc64
378535 Valgrind reports INTERNAL ERROR in execve syscall wrapper
378673 Update libiberty demangler
378931 Add ISA 3.0B additional isnstructions, add OV32, CA32 setting support
379039 syscall wrapper for prctl(PR_SET_NAME) must not check more than 16 bytes
379094 Valgrind reports INTERNAL ERROR in rt_sigsuspend syscall wrapper
379371 UNKNOWN task message [id 3444, to mach_task_self(), reply 0x603]
(task_register_dyld_image_infos)
379372 UNKNOWN task message [id 3447, to mach_task_self(), reply 0x603]
(task_register_dyld_shared_cache_image_info)
379390 unhandled syscall: mach:70 (host_create_mach_voucher_trap)
379473 MIPS: add support for rdhwr cycle counter register
379504 remove TileGX/Linux port
379525 Support more x86 nop opcodes
379838 disAMode(x86): not an addr!
379703 PC ISA 3.0 fixes: stxvx, stxv, xscmpexpdp instructions
379890 arm: unhandled instruction: 0xEBAD 0x1B05 (sub.w fp, sp, r5, lsl #4)
379895 clock_gettime does not execute POST syscall wrapper
379925 PPC64, mtffs does not set the FPCC and C bits in the FPSCR correctly
379966 WARNING: unhandled amd64-linux syscall: 313 (finit_module)
380200 xtree generated callgrind files refer to files without directory name
380202 Assertion failure for cache line size (cls == 64) on aarch64.
380397 s390x: __GI_strcspn() replacement needed
n-i-bz Fix pub_tool_basics.h build issue with g++ 4.4.7.

(3.13.0.RC1: 2 June 2017, vex r3386, valgrind r16434)
(3.13.0.RC2: 9 June 2017, vex r3389, valgrind r16443)
(3.13.0: 14 June 2017, vex r3396, valgrind r16446)

Release 3.12.0 (20 October 2016)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.12.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, X86/MacOSX 10.10 and AMD64/MacOSX 10.10. There is also preliminary support for X86/MacOSX 10.11/12, AMD64/MacOSX 10.11/12 and TILEGX/Linux.

* ================== PLATFORM CHANGES =================

* POWER: Support for ISA 3.0 has been added

* mips: support for O32 FPXX ABI has been added.
* mips: improved recognition of different processors
* mips: determination of page size now done at run time

* amd64: Partial support for AMD FMA4 instructions.

* arm, arm64: Support for v8 crypto and CRC instructions.

* Improvements and robustification of the Solaris port.

* Preliminary support for MacOS 10.12 (Sierra) has been added.

Whilst 3.12.0 continues to support the 32-bit x86 instruction set, we would prefer users to migrate to 64-bit x86 (a.k.a amd64 or x86_64) where possible. Valgrind's support for 32-bit x86 has stagnated in recent years and has fallen far behind that for 64-bit x86 instructions. By contrast 64-bit x86 is well supported, up to and including AVX2.

* ==================== TOOL CHANGES ====================

* Memcheck:

  - Added meta mempool support for describing a custom allocator which:
    - Auto-frees all chunks assuming that destroying a pool destroys all objects in the pool
    - Uses itself to allocate other memory blocks

  - New flag --ignore-range-below-sp to ignore memory accesses below the stack pointer, if you really have to. The related flag --workaround-gcc296-bugs=yes is now deprecated. Use --ignore-range-below-sp=1024-1 as a replacement.

* DRD:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- Improved thread startup time significantly on non-Linux platforms.

* DHAT

- Added collection of the metric "tot-blocks-allocd"

* ==================== OTHER CHANGES ====================

* Replacement/wrapping of malloc/new related functions is now done not just for system libraries by default, but for any globally defined malloc/new related function (both in shared libraries and statically linked alternative malloc implementations). The dynamic (runtime) linker is excluded, though. To only intercept malloc/new related functions in system libraries use --soname-synonyms=somalloc=nouserintercepts (where "nouserintercepts" can be any non-existing library name). This new functionality is not implemented for MacOS X.

* The maximum number of callers in a suppression entry is now equal to the maximum size for --num-callers (500). Note that --gen-suppressions=yes|all similarly generates suppressions containing up to --num-callers frames.

* New and modified GDB server monitor features:

- Valgrind's gdbserver now accepts the command 'catch syscall'. Note that you must have GDB >= 7.11 to use 'catch syscall' with gdbserver.

* New option --run-cxx-freeres=<yes|no> can be used to change whether __gnu_cxx::__freeres() cleanup function is called or not. Default is 'yes'.

* Valgrind is able to read compressed debuginfo sections in two formats:
- zlib ELF gABI format with SHF_COMPRESSED flag (gcc option -gz=zlib)
- zlib GNU format with .zdebug sections (gcc option -gz=zlib-gnu)

* Modest JIT-cost improvements: the cost of instrumenting code blocks for the most common use case (x86_64-linux, Memcheck) has been reduced by 10%-15%.

* Improved performance for programs that do a lot of discarding of instruction address ranges of 8KB or less.

* The C++ symbol demangler has been updated.

* More robustness against invalid syscall parameters on Linux.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather

than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

```
191069  Exiting due to signal not reported in XML output
199468  Suppressions: stack size limited to 25
        while --num-callers allows more frames
212352  vex amd64 unhandled opc_aux = 0x 2, first_opcode == 0xDC (FCOM)
278744  cvtps2pd with redundant RexW
303877  valgrind doesn't support compressed debuginfo sections.
345307  Warning about "still reachable" memory when using libstdc++ from gcc 5
348345  Assertion fails for negative lineno
348924  MIPS: Load doubles through memory so the code compiles with the FPXX ABI
351282  V 3.10.1 MIPS softfloat build broken with GCC 4.9.3 / binutils 2.25.1
351692  Dumps created by valgrind are not readable by gdb (mips32 specific)
351804  Crash on generating suppressions for "printf" call on OS X 10.10
352197  mips: mmap2() not wrapped correctly for page size > 4096
353083  arm64 doesn't implement various xattr system calls
353084  arm64 doesn't support sigpending system call
353137  www: update info for Supported Platforms
353138  www: update "The Valgrind Developers" page
353370  don't advertise RDRAND in cpuid for Core-i7-4910-like avx2 machine
        == 365325
        == 357873
353384  amd64->IR: 0x66 0xF 0x3A 0x62 0xD1 0x62 (pcmpXstrX $0x62)
353398  WARNING: unhandled amd64-solaris syscall: 207
353660  XML in auxwhat tag not escaping reserved symbols properly
353680  s390x: Crash with certain glibc versions due to non-implemented TBEGIN
353727  amd64->IR: 0x66 0xF 0x3A 0x62 0xD1 0x72 (pcmpXstrX $0x72)
353802  ELF debug info reader confused with multiple .rodata sections
353891  Assert 'bad_scanned_addr < VG_ROUNDDN(start+len, sizeof(Addr))' failed
353917  unhandled amd64-solaris syscall fchdir(120)
353920  unhandled amd64-solaris syscall: 170
354274  arm: unhandled instruction: 0xEBAD 0x0AC1 (sub.w sl, sp, r1, lsl #3)
354392  unhandled amd64-solaris syscall: 171
354797  Vbit test does not include Iops for Power 8 instruction support
354883  tst->os_state.pthread - magic_delta assertion failure on OSX 10.11
        == 361351
        == 362920
        == 366222
354933  Fix documentation of --kernel-variant=android-no-hw-tls option
355188  valgrind should intercept all malloc related global functions
355454  do not intercept malloc related symbols from the runtime linker
355455  stderr.exp of test cases wrapmalloc and wrapmallocstatic overconstrained
356044  Dwarf line info reader misinterprets is_stmt register
356112  mips: replace addi with addiu
356393  valgrind (vex) crashes because isZeroU happened
        == 363497
        == 364497
356676  arm64-linux: unhandled syscalls 125, 126 (sched_get_priority_max/min)
356678  arm64-linux: unhandled syscall 232 (mincore)
```

356817 valgrind.h triggers compiler errors on MSVC when defining NVALGRIND
356823 Unsupported ARM instruction: stlex
357059 x86/amd64: SSE cvtpi2ps with memory source does transition to MMX state
357338 Unhandled instruction for SHA instructions libcrypto Boring SSL
357673 crash if I try to run valgrind with a binary link with libcurl
357833 Setting RLIMIT_DATA to zero breaks with linux 4.5+
357871 pthread_spin_destroy not properly wrapped
357887 Calls to VG_(fclose) do not close the file descriptor
357932 amd64->IR: accept redundant REX prefixes for {minsd,maxsd} m128, xmm.
358030 support direct socket calls on x86 32bit (new in linux 4.3)
358478 drd/tests/std_thread.cpp doesn't build with GCC6
359133 Assertion 'eltSzB <= ddpa->poolSzB' failed
359181 Buffer Overflow during Demangling
359201 futex syscall "skips" argument 5 if op is FUTEX_WAIT_BITSET
359289 s390x: popcnt (B9E1) not implemented
359472 The Power PC vsubuqm instruction doesn't always give the correct result
359503 Add missing syscalls for aarch64 (arm64)
359645 "You need libc6-dbg" help message could be more helpful
359703 s390: wire up separate socketcalls system calls
359724 getsockname might crash - deref_UInt should call safe_to_deref
359733 amd64 implement ld.so strchr/index override like x86
359767 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 1/5
359829 Power PC test suite none/tests/ppc64/test_isa_2_07.c uses
uninitialized data
359838 arm64: Unhandled instruction 0xD5033F5F (clrex)
359871 Incorrect mask handling in ppoll
359952 Unrecognised PCMPESTRM variants (0x70, 0x19)
360008 Contents of Power vr registers contents is not printed correctly when
the --vgdb-shadow-registers=yes option is used
360035 POWER PC instruction bcdadd and bcdsubtract generate result with
non-zero shadow bits
360378 arm64: Unhandled instruction 0x5E280844 (sha1h s4, s2)
360425 arm64 unsupported instruction ldpsw
== 364435
360519 none/tests/arm64/memory.vgtest might fail with newer gcc
360571 Error about the Android Runtime reading below the stack pointer on ARM
360574 Wrong parameter type for an ashmem ioctl() call on Android and ARM64
360749 kludge for multiple .rodata sections on Solaris no longer needed
360752 raise the number of reserved fds in m_main.c from 10 to 12
361207 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 2/5
361226 s390x: risbgn (EC59) not implemented
361253 [s390x] ex_clone.c:42: undefined reference to `pthread_create'
361354 ppc64[le]: wire up separate socketcalls system calls
361615 Inconsistent termination for multithreaded process terminated by signal
361926 Unhandled Solaris syscall: sysfs(84)
362009 V dumps core on unimplemented functionality before threads are created
362329 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 3/5
362894 missing (broken) support for wbit field on mtfsfi instruction (ppc64)
362935 [AsusWRT] Assertion 'sizeof(TTEntryC) <= 88' failed
362953 Request for an update to the Valgrind Developers page
363680 add renameat2() support
363705 arm64 missing syscall name_to_handle_at and open_by_handle_at
363714 ppc64 missing syscalls sync, waitid and name_to/open_by_handle_at
363858 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 4/5

364058 clarify in manual limitations of array overruns detections
364413 pselect sycallwrapper mishandles NULL sigmask
364728 Power PC, missing support for several HW registers in
get_otrack_shadow_offset_wrk()
364948 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 5/5
365273 Invalid write to stack location reported after signal handler runs
365912 ppc64BE segfault during jm-insns test (RELRO)
366079 FPXX Support for MIPS32 Valgrind
366138 Fix configure errors out when using Xcode 8 (clang 8.0.0)
366344 Multiple unhandled instruction for Aarch64
(0x0EE0E020, 0x1AC15800, 0x4E284801, 0x5E040023, 0x5E056060)
367995 Integration of memcheck with custom memory allocator
368120 x86_linux asm _start functions do not keep 16-byte aligned stack pointer
368412 False positive result for altivec capability check
368416 Add tc06_two_races_xml.exp output for ppc64
368419 Perf Events ioctls not implemented
368461 mmapunmap test fails on ppc64
368823 run_a_thread_NORETURN assembly code typo for VGP_arm64_linux target
369000 AMD64 fma4 instructions unsupported.
369169 ppc64 fails jm_int_isa_2_07 test
369175 jm_vec_isa_2_07 test crashes on ppc64
369209 valgrind loops and eats up all memory if cwd doesn't exist.
369356 pre_mem_read_sockaddr syscall wrapper can crash with bad sockaddr
369359 msghdr_foreachfield can crash when handling bad iovec
369360 Bad sigprocmask old or new sets can crash valgrind
369361 vmsplice syscall wrapper crashes on bad iovec
369362 Bad sigaction arguments crash valgrind
369383 x86 sys_modify_ldt wrapper crashes on bad ptr
369402 Bad set/get_thread_area pointer crashes valgrind
369441 bad lvec argument crashes process_vm_readv/writev syscall wrappers
369446 valgrind crashes on unknown fcntl command
369439 S390x: Unhandled insns RISBLG/RISBHG and LDE/LDER
369468 Remove quadratic metapool algorithm using VG_(HT_remove_at_Iter)
370265 ISA 3.0 HW cap stuff needs updating
371128 BCD add and subtract instructions on Power BE in 32-bit mode do not work
372195 Power PC, xxsel instruction is not always recognized

n-i-bz Fix incorrect (or infinite loop) unwind on RHEL7 x86 and amd64
n-i-bz massif --pages-as-heap=yes does not report peak caused by mmap+munmap
n-i-bz false positive leaks due to aspacemgr merging heap & non heap segments
n-i-bz Fix ppoll_alarm exclusion on OS X
n-i-bz Document brk segment limitation, reference manual in limit reached msg.
n-i-bz Fix clobber list in none/tests/amd64/xacq_xrel.c [valgrind r15737]
n-i-bz Bump allowed shift value for "add.w reg, sp, reg, lsl #N" [vex r3206]
n-i-bz amd64: memcheck false positive with shr %edx
n-i-bz arm3: Allow early writeback of SP base register in "strd rD, [sp, #-16]"
n-i-bz ppc: Fix two cases of PPCAvFpOp vs PPCFpOp enum confusion
n-i-bz arm: Fix incorrect register-number constraint check for LDAEX{,B,H,D}
n-i-bz DHAT: added collection of the metric "tot-blocks-allocd"

(3.12.0.RC1: 20 October 2016, vex r3282, valgrind r16094)
(3.12.0.RC2: 20 October 2016, vex r3282, valgrind r16096)
(3.12.0: 21 October 2016, vex r3282, valgrind r16098)

Release 3.11.0 (22 September 2015)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.11.0 is a feature release with many improvements and the usual
collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux,
ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux,
MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android,
MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, X86/MacOSX
10.10 and AMD64/MacOSX 10.10. There is also preliminary support for
X86/MacOSX 10.11, AMD64/MacOSX 10.11 and TILEGX/Linux.

* ================== PLATFORM CHANGES =================

* Support for Solaris/x86 and Solaris/amd64 has been added.

* Preliminary support for Mac OS X 10.11 (El Capitan) has been added.

* Preliminary support for the Tilera TileGX architecture has been added.

* s390x: It is now required for the host to have the "long displacement"
  facility. The oldest supported machine model is z990.

* x86: on an SSE2 only host, Valgrind in 32 bit mode now claims to be a
  Pentium 4. 3.10.1 wrongly claimed to be a Core 2, which is SSSE3.

* The JIT's register allocator is significantly faster, making the JIT
  as a whole somewhat faster, so JIT-intensive activities, for example
  program startup, are modestly faster, around 5%.

* There have been changes to the default settings of several command
  line flags, as detailed below.

* Intel AVX2 support is more complete (64 bit targets only). On AVX2
  capable hosts, the simulated CPUID will now indicate AVX2 support.

* ==================== TOOL CHANGES ====================

* Memcheck:

  - The default value for --leak-check-heuristics has been changed from
    "none" to "all". This helps to reduce the number of possibly
    lost blocks, in particular for C++ applications.

  - The default value for --keep-stacktraces has been changed from
    "malloc-then-free" to "malloc-and-free". This has a small cost in
    memory (one word per malloc-ed block) but allows Memcheck to show the
    3 stacktraces of a dangling reference: where the block was allocated,
    where it was freed, and where it is acccessed after being freed.

  - The default value for --partial-loads-ok has been changed from "no" to
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

"yes", so as to avoid false positive errors resulting from some kinds of vectorised loops.

- A new monitor command 'xb <addr> <len>' shows the validity bits of <len> bytes at <addr>. The monitor command 'xb' is easier to use than get_vbits when you need to associate byte data value with their corresponding validity bits.

- The 'block_list' monitor command has been enhanced:
  o it can print a range of loss records
  o it now accepts an optional argument 'limited <max_blocks>' to control the number of blocks printed.
  o if a block has been found using a heuristic, then 'block_list' now shows the heuristic after the block size.
  o the loss records/blocks to print can be limited to the blocks found via specified heuristics.

- The C helper functions used to instrument loads on x86-{linux,solaris} and arm-linux (both 32-bit only) have been replaced by handwritten assembly sequences. This gives speedups in the region of 0% to 7% for those targets only.

- A new command line option, --expensive-definedness-checks=yes|no, has been added. This is useful for avoiding occasional invalid uninitialised-value errors in optimised code. Watch out for runtime degradation, as this can be up to 25%. As always, though, the slowdown is highly application specific. The default setting is "no".

* Massif:

- A new monitor command 'all_snapshots <filename>' dumps all snapshots taken so far.

* Helgrind:

- Significant memory reduction and moderate speedups for --history-level=full for applications accessing a lot of memory with many different stacktraces.

- The default value for --conflict-cache-size=N has been doubled to 2000000. Users that were not using the default value should preferably also double the value they give.

  The default was changed due to the changes in the "full history" implementation. Doubling the value gives on average a slightly more complete history and uses similar memory (or significantly less memory in the worst case) than the previous implementation.

- The Helgrind monitor command 'info locks' now accepts an optional argument 'lock_addr', which shows information about the lock at the given address only.

- When using --history-level=full, the new Helgrind monitor command

'accesshistory <addr> [<len>]' will show the recorded accesses for
<len> (or 1) bytes at <addr>.

* =================== OTHER CHANGES ===================

* The default value for the --smc-check option has been changed from "stack" to "all-non-file" on targets that provide automatic D-I cache coherence (x86, amd64 and s390x). The result is to provide, by default, transparent support for JIT generated and self-modifying code on all targets.

* Mac OS X only: the default value for the --dsymutil option has been changed from "no" to "yes", since any serious usage on Mac OS X always required it to be "yes".

* The command line options --db-attach and --db-command have been removed. They were deprecated in 3.10.0.

* When a process dies due to a signal, Valgrind now shows the signal and the stacktrace at default verbosity (i.e. verbosity 1).

* The address description logic used by Memcheck and Helgrind now describes addresses in anonymous segments, file mmap-ed segments, shared memory segments and the brk data segment.

* The new option --error-markers=<begin>,<end> can be used to mark the begin/end of errors in textual output mode, to facilitate searching/extracting errors in output files that mix valgrind errors with program output.

* The new option --max-threads=<number> can be used to change the number of threads valgrind can handle. The default is 500 threads which should be more than enough for most applications.

* The new option --valgrind-stacksize=<number> can be used to change the size of the private thread stacks used by Valgrind. This is useful for reducing memory use or increasing the stack size if Valgrind segfaults due to stack overflow.

* The new option --avg-transtab-entry-size=<number> can be used to specify the expected instrumented block size, either to reduce memory use or to avoid excessive retranslation.

* Valgrind can be built with Intel's ICC compiler, version 14.0 or later.

* New and modified GDB server monitor features:

  - When a signal is reported in GDB, you can now use the GDB convenience variable $_siginfo to examine detailed signal information.

  - Valgrind's gdbserver now allows the user to change the signal to deliver to the process. So, use 'signal SIGNAL' to continue execution with SIGNAL instead of the signal reported to GDB. Use 'signal 0' to continue without passing the signal to the process.

```
  - With GDB >= 7.10, the command 'target remote'
    will automatically load the executable file of the process running
    under Valgrind. This means you do not need to specify the executable
    file yourself, GDB will discover it itself.  See GDB documentation about
    'qXfer:exec-file:read' packet for more info.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved.  Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry.  We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

116002  VG_(printf): Problems with justification of strings and integers
155125  avoid cutting away file:lineno after long function name
197259  Unsupported arch_prtctl PR_SET_GS option
201152  ppc64: Assertion in ppc32g_dirtyhelper_MFSPR_268_269
201216  Fix Valgrind does not support pthread_sigmask() on OS X
201435  Fix Darwin: -v does not show kernel version
208217  "Warning: noted but unhandled ioctl 0x2000747b" on Mac OS X
211256  Fixed an outdated comment regarding the default platform.
211529  Incomplete call stacks for code compiled by newer versions of MSVC
211926  Avoid compilation warnings in valgrind.h with -pedantic
212291  Fix unhandled syscall: unix:132 (mkfifo) on OS X
        == 263119
226609  Crediting upstream authors in man page
231257  Valgrind omits path when executing script from shebang line
254164  OS X task_info: UNKNOWN task message [id 3405, to mach_task_self() [..]
294065  Improve the pdb file reader by avoiding hardwired absolute pathnames
269360  s390x: Fix addressing mode selection for compare-and-swap
302630  Memcheck: Assertion failed: 'sizeof(UWord) == sizeof(UInt)'
        == 326797
312989  ioctl handling needs to do POST handling on generic ioctls and [..]
319274  Fix unhandled syscall: unix:410 (sigsuspend_nocancel) on OS X
324181  mmap does not handle MAP_32BIT (handle it now, rather than fail it)
327745  Fix valgrind 3.9.0 build fails on Mac OS X 10.6.8
330147  libmpiwrap PMPI_Get_count returns undefined value
333051  mmap of huge pages fails due to incorrect alignment
        == 339163
334802  valgrind does not always explain why a given option is bad
335618  mov.w rN, pc/sp (ARM32)
335785  amd64->IR 0xC4 0xE2 0x75 0x2F (vmaskmovpd)
        == 307399
        == 343175
        == 342740
        == 346912
335907  segfault when running wine's ddrawex/tests/surface.c under valgrind
```

338602 AVX2 bit in CPUID missing
338606 Strange message for scripts with invalid interpreter
338731 ppc: Fix testuite build for toolchains not supporting -maltivec
338995 shmat with hugepages (SHM_HUGETLB) fails with EINVAL
339045 Getting valgrind to compile and run on OS X Yosemite (10.10)
== 340252
339156 gdbsrv not called for fatal signal
339215 Valgrind 3.10.0 contain 2013 in copyrights notice
339288 support Cavium Octeon MIPS specific BBIT*32 instructions
339636 Use fxsave64 and fxrstor64 mnemonics instead of old-school rex64 prefix
339442 Fix testsuite build failure on OS X 10.9
339542 Enable compilation with Intel's ICC compiler
339563 The DVB demux DMX_STOP ioctl doesn't have a wrapper
339688 Mac-specific ASM does not support .version directive (cpuid,
tronical and pushfpopf tests)
339745 Valgrind crash when check Marmalade app (partial fix)
339755 Fix known deliberate memory leak in setenv() on Mac OS X 10.9
339778 Linux/TileGx platform support for Valgrind
339780 Fix known uninitialised read in pthread_rwlock_init() on Mac OS X 10.9
339789 Fix none/tests/execve test on Mac OS X 10.9
339808 Fix none/tests/rlimit64_nofile test on Mac OS X 10.9
339820 vex amd64->IR: 0x66 0xF 0x3A 0x63 0xA 0x42 0x74 0x9 (pcmpistri $0x42)
340115 Fix none/tests/cmdline[1|2] tests on systems which define TMPDIR
340392 Allow user to select more accurate definedness checking in memcheck
to avoid invalid complaints on optimised code
340430 Fix some grammatical weirdness in the manual.
341238 Recognize GCC5/DWARFv5 DW_LANG constants (Go, C11, C++11, C++14)
341419 Signal handler ucontext_t not filled out correctly on OS X
341539 VG_(describe_addr) should not describe address as belonging to client
segment if it is past the heap end
341613 Enable building of manythreads and thread-exits tests on Mac OS X
341615 Fix none/tests/darwin/access_extended test on Mac OS X
341698 Valgrind's AESKEYGENASSIST gives wrong result in words 0 and 2 [..]
341789 aarch64: shmat fails with valgrind on ARMv8
341997 MIPS64: Cavium OCTEON insns - immediate operand handled incorrectly
342008 valgrind.h needs type cast [..] for clang/llvm in 64-bit mode
342038 Unhandled syscalls on aarch64 (mbind/get/set_mempolicy)
342063 wrong format specifier for test mcblocklistsearch in gdbserver_tests
342117 Hang when loading PDB file for MSVC compiled Firefox under Wine
342221 socket connect false positive uninit memory for unknown af family
342353 Allow dumping full massif output while valgrind is still running
342571 Valgrind chokes on AVX compare intrinsic with _CMP_GE_QS
== 346476
== 348387
== 350593
342603 Add I2C_SMBUS ioctl support
342635 OS X 10.10 (Yosemite) - missing system calls and fcntl code
342683 Mark memory past the initial brk limit as unaddressable
342783 arm: unhandled instruction 0xEEFE1ACA = "vcvt.s32.f32 s3, s3, #12"
342795 Internal glibc __GI_mempcpy call should be intercepted
342841 s390x: Support instructions fiebr(a) and fidbr(a)
343012 Unhandled syscall 319 (memfd_create)
343069 Patch updating v4l2 API support
343173 helgrind crash during stack unwind

343219 fix GET_STARTREGS for arm
343303 Fix known deliberate memory leak in setenv() on Mac OS X 10.10
343306 OS X 10.10: UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option
343332 Unhandled instruction 0x9E310021 (fcvtmu) on aarch64
343335 unhandled instruction 0x1E638400 (fccmp) aarch64
343523 OS X mach_ports_register: UNKNOWN task message [id 3403, to [..]
343525 OS X host_get_special_port: UNKNOWN host message [id 412, to [..]
343597 ppc64le: incorrect use of offseof macro
343649 OS X host_create_mach_voucher: UNKNOWN host message [id 222, to [..]
343663 OS X 10.10 Memchecj always reports a leak regardless of [..]
343732 Unhandled syscall 144 (setgid) on aarch64
343733 Unhandled syscall 187 (msgctl and related) on aarch64
343802 s390x: False positive "conditional jump or move depends on [..]
343902 --vgdb=yes doesn't break when --xml=yes is used
343967 Don't warn about setuid/setgid/setcap executable for directories
343978 Recognize DWARF5/GCC5 DW_LANG_Fortran 2003 and 2008 constants
344007 accept4 syscall unhandled on arm64 (242) and ppc64 (344)
344033 Helgrind on ARM32 loses track of mutex state in pthread_cond_wait
344054 www - update info for Solaris/illumos
344416 'make regtest' does not work cleanly on OS X
344235 Remove duplicate include of pub_core_aspacemgr.h
344279 syscall sendmmsg on arm64 (269) and ppc32/64 (349) unhandled
344295 syscall recvmmsg on arm64 (243) and ppc32/64 (343) unhandled
344307 2 unhandled syscalls on aarch64/arm64: umount2(39), mount (40)
344314 callgrind_annotate ... warnings about commands containing newlines
344318 socketcall should wrap recvmmsg and sendmmsg
344337 Fix unhandled syscall: mach:41 (_kernelrpc_mach_port_guard_trap)
344416 Fix 'make regtest' does not work cleanly on OS X
344499 Fix compilation for Linux kernel >= 4.0.0
344512 OS X: unhandled syscall: unix:348 (__pthread_chdir),
unix:349 (__pthread_fchdir)
344559 Garbage collection of unused segment names in address space manager
344560 Fix stack traces missing penultimate frame on OS X
344621 Fix memcheck/tests/err_disable4 test on OS X
344686 Fix suppression for pthread_rwlock_init on OS X 10.10
344702 Fix missing libobjc suppressions on OS X 10.10
== 344543
344936 Fix unhandled syscall: unix:473 (readlinkat) on OS X 10.10
344939 Fix memcheck/tests/xml1 on OS X 10.10
345016 helgrind/tests/locked_vs_unlocked2 is failing sometimes
345079 Fix build problems in VEX/useful/test_main.c
345126 Incorrect handling of VIDIOC_G_AUDIO and G_AUDOUT
345177 arm64: prfm (reg) not implemented
345215 Performance improvements for the register allocator
345248 add support for Solaris OS in valgrind
345338 TIOCGSERIAL and TIOCSSERIAL ioctl support on Linux
345394 Fix memcheck/tests/strchr on OS X
345637 Fix memcheck/tests/sendmsg on OS X
345695 Add POWERPC support for AT_DCACHESIZE and HWCAP2
345824 Fix aspacem segment mismatch: seen with none/tests/bigcode
345887 Fix an assertion in the address space manager
345928 amd64: callstack only contains current function for small stacks
345984 disInstr(arm): unhandled instruction: 0xEE193F1E
345987 MIPS64: Implement cavium LHX instruction

346031 MIPS: Implement support for the CvmCount register (rhwr %0, 31)
346185 Fix typo saving altivec register v24
346267 Compiler warnings for PPC64 code on call to LibVEX_GuestPPC64_get_XER()
and LibVEX_GuestPPC64_get_CR()
346270 Regression tests none/tests/jm_vec/isa_2_07 and
none/tests/test_isa_2_07_part2 have failures on PPC64 little endian
346307 fuse filesystem syscall deadlocks
346324 PPC64 missing support for lbarx, lharx, stbcx and sthcx instructions
346411 MIPS: SysRes::_valEx handling is incorrect
346416 Add support for LL_IOC_PATH2FID and LL_IOC_GETPARENT Lustre ioctls
346474 PPC64 Power 8, spr TEXASRU register not supported
346487 Compiler generates "note" about a future ABI change for PPC64
346562 MIPS64: lwl/lwr instructions are performing 64bit loads
and causing spurious "invalid read of size 8" warnings
346801 Fix link error on OS X: _vgModuleLocal_sf_maybe_extend_stack
347151 Fix suppression for pthread_rwlock_init on OS X 10.8
347233 Fix memcheck/tests/strchr on OS X 10.10 (Haswell)
347322 Power PC regression test cleanup
347379 valgrind --leak-check=full leak errors from system libs on OS X 10.8
== 217236
347389 unhandled syscall: 373 (Linux ARM syncfs)
347686 Patch set to cleanup PPC64 regtests
347978 Remove bash dependencies where not needed
347982 OS X: undefined symbols for architecture x86_64: "_global" [..]
347988 Memcheck: the 'impossible' happened: unexpected size for Addr (OSX/wine)
== 345929
348102 Patch updating v4l2 API support
348247 amd64 front end: jno jumps wrongly when overflow is not set
348269 Improve mmap MAP_HUGETLB support.
348334 (ppc) valgrind does not simulate dcbfl - then my program terminates
348345 Assertion fails for negative lineno
348377 Unsupported ARM instruction: yield
348565 Fix detection of command line option availability for clang
348574 vex amd64->IR pcmpistri SSE4.2 unsupported (pcmpistri $0x18)
348728 Fix broken check for VIDIOC_G_ENC_INDEX
348748 Fix redundant condition
348890 Fix clang warning about unsupported --param inline-unit-growth=900
348949 Bogus "ERROR: --ignore-ranges: suspiciously large range"
349034 Add Lustre ioctls LL_IOC_GROUP_LOCK and LL_IOC_GROUP_UNLOCK
349086 Fix UNKNOWN task message [id 3406, to mach_task_self(), [..]
349087 Fix UNKNOWN task message [id 3410, to mach_task_self(), [..]
349626 Implemented additional Xen hypercalls
349769 Clang/osx: ld: warning: -read_only_relocs cannot be used with x86_64
349790 Clean up of the hardware capability checking utilities.
349828 memcpy intercepts memmove causing src/dst overlap error (ppc64 ld.so)
349874 Fix typos in source code
349879 memcheck: add handwritten assembly for helperc_LOADV*
349941 di_notify_mmap might create wrong start/size DebugInfoMapping
350062 vex x86->IR: 0x66 0xF 0x3A 0xB (ROUNDSD) on OS X
350202 Add limited param to 'monitor block_list'
350290 s390x: Support instructions fixbr(a)
350359 memcheck/tests/x86/fxsave hangs indefinetely on OS X
350809 Fix none/tests/async-sigs for Solaris
350811 Remove reference to --db-attach which has been removed.

350813 Memcheck/x86: enable handwritten assembly helpers for x86/Solaris too
350854 hard-to-understand code in VG_(load_ELF)()
351140 arm64 syscalls setuid (146) and setresgid (149) not implemented
351386 Solaris: Cannot run ld.so.1 under Valgrind
351474 Fix VG_(iseqsigset) as obvious
351531 Typo in /include/vki/vki-xen-physdev.h header guard
351756 Intercept platform_memchr$VARIANT$Haswell on OS X
351858 ldsoexec support on Solaris
351873 Newer gcc doesn't allow __builtin_tabortdc[i] in ppc32 mode
352130 helgrind reports false races for printfs using mempcpy on FILE* state
352284 s390: Conditional jump depends on uninitialised value(s) in vfprintf
352320 arm64 crash on none/tests/nestedfs
352765 Vbit test fails on Power 6
352768 The mbar instruction is missing from the Power PC support
352769 Power PC program priority register (PPR) is not supported
n-i-bz Provide implementations of certain compiler builtins to support
        compilers that may not provide those
n-i-bz Old STABS code is still being compiled, but never used. Remove it.
n-i-bz Fix compilation on distros with glibc < 2.5
n-i-bz (vex 3098) Avoid generation of Neon insns on non-Neon hosts
n-i-bz Enable rt_sigpending syscall on ppc64 linux.
n-i-bz mremap did not work properly on shared memory
n-i-bz Fix incorrect sizeof expression in syswrap-xen.c reported by Coverity
n-i-bz In VALGRIND_PRINTF write out thread name, if any, to xml

(3.11.0.TEST1: 8 September 2015, vex r3187, valgrind r15646)
(3.11.0.TEST2: 21 September 2015, vex r3193, valgrind r15667)
(3.11.0:   22 September 2015, vex r3195, valgrind r15674)

Release 3.10.1 (25 November 2014)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3.10.1 is a bug fix release. It fixes various bugs reported in 3.10.0
and backports fixes for all reported missing AArch64 ARMv8 instructions
and syscalls from the trunk. If you package or deliver 3.10.0 for others
to use, you might want to consider upgrading to 3.10.1 instead.

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

335440 arm64: ld1 (single structure) is not implemented
335713 arm64: unhanded instruction: prfm (immediate)
339020 ppc64: memcheck/tests/ppc64/power_ISA2_05 failing in nightly build
339182 ppc64: AvSplat ought to load destination vector register with [..]
339336 PPC64 store quad instruction (stq) is not supposed to change [..]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

339433 ppc64 lxvw4x instruction uses four 32-byte loads
339645 Use correct tag names in sys_getdents/64 wrappers
339706 Fix false positive for ioctl(TIOCSIG) on linux
339721 assertion 'check_sibling == sibling' failed in readdwarf3.c ...
339853 arm64 times syscall unknown
339855 arm64 unhandled getsid/setsid syscalls
339858 arm64 dmb sy not implemented
339926 Unhandled instruction 0x1E674001 (frintx) on aarm64
339927 Unhandled instruction 0x9E7100C6 (fcvtmu) on aarch64
339938 disInstr(arm64): unhandled instruction 0x4F8010A4 (fmla)
        == 339950
339940 arm64: unhandled syscall: 83 (sys_fdatasync) + patch
340033 arm64: unhandled insn dmb ishld and some other isb-dmb-dsb variants
340028 unhandled syscalls for arm64 (msync, pread64, setreuid and setregid)
340036 arm64: Unhandled instruction ld4 (multiple structures, no offset)
340236 arm64: unhandled syscalls: mknodat, fchdir, chroot, fchownat
340509 arm64: unhandled instruction fcvtas
340630 arm64: fchmod (52) and fchown (55) syscalls not recognized
340632 arm64: unhandled instruction fcvtas
340722 Resolve "UNKNOWN attrlist flags 0:0x10000000"
340725 AVX2: Incorrect decoding of vpbroadcast{b,w} reg,reg forms
340788 warning: unhandled syscall: 318 (getrandom)
340807 disInstr(arm): unhandled instruction: 0xEE989B20
340856 disInstr(arm64): unhandled instruction 0x1E634C45 (fcsel)
340922 arm64: unhandled getgroups/setgroups syscalls
350251 Fix typo in VEX utility program (test_main.c).
350407 arm64: unhandled instruction ucvtf (vector, integer)
350809 none/tests/async-sigs breaks when run under cron on Solaris
350811 update README.solaris after r15445
350813 Use handwritten memcheck assembly helpers on x86/Solaris [..]
350854 strange code in VG_(load_ELF)()
351140 arm64 syscalls setuid (146) and setresgid (149) not implemented
n-i-bz DRD and Helgrind: Handle Imbe_CancelReservation (clrex on ARM)
n-i-bz Add missing]] to terminate CDATA.
n-i-bz Glibc versions prior to 2.5 do not define PTRACE_GETSIGINFO
n-i-bz Enable sys_fadvise64_64 on arm32.
n-i-bz Add test cases for all remaining AArch64 SIMD, FP and memory insns.
n-i-bz Add test cases for all known arm64 load/store instructions.
n-i-bz PRE(sys_openat): when checking whether ARG1 == VKI_AT_FDCWD [..]
n-i-bz Add detection of old ppc32 magic instructions from bug 278808.
n-i-bz exp-dhat: Implement missing function "dh_malloc_usable_size".
n-i-bz arm64: Implement "fcvtpu w, s".
n-i-bz arm64: implement ADDP and various others
n-i-bz arm64: Implement {S,U}CVTF (scalar, fixedpt).
n-i-bz arm64: enable FCVT{A,N}S X,S.

(3.10.1: 25 November 2014, vex r3026, valgrind r14785)

Release 3.10.0 (10 September 2014)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.10.0 is a feature release with many improvements and the usual
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, MIPS32/Android, X86/Android, X86/MacOSX 10.9 and AMD64/MacOSX 10.9. Support for MacOSX 10.8 and 10.9 is significantly improved relative to the 3.9.0 release.

* ================== PLATFORM CHANGES =================

* Support for the 64-bit ARM Architecture (AArch64 ARMv8). This port is mostly complete, and is usable, but some SIMD instructions are as yet unsupported.

* Support for little-endian variant of the 64-bit POWER architecture.

* Support for Android on MIPS32.

* Support for 64bit FPU on MIPS32 platforms.

* Both 32- and 64-bit executables are supported on MacOSX 10.8 and 10.9.

* Configuration for and running on Android targets has changed. See README.android in the source tree for details.

* ================== DEPRECATED FEATURES =================

* --db-attach is now deprecated and will be removed in the next valgrind feature release. The built-in GDB server capabilities are superior and should be used instead. Learn more here: http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver

* ==================== TOOL CHANGES ====================

* Memcheck:

  - Client code can now selectively disable and re-enable reporting of invalid address errors in specific ranges using the new client requests VALGRIND_DISABLE_ADDR_ERROR_REPORTING_IN_RANGE and VALGRIND_ENABLE_ADDR_ERROR_REPORTING_IN_RANGE.

  - Leak checker: there is a new leak check heuristic called "length64". This is used to detect interior pointers pointing 8 bytes inside a block, on the assumption that the first 8 bytes holds the value "block size - 8". This is used by sqlite3MemMalloc, for example.

  - Checking of system call parameters: if a syscall parameter (e.g. bind struct sockaddr, sendmsg struct msghdr, ...) has several fields not initialised, an error is now reported for each field. Previously, an error was reported only for the first uninitialised field.

  - Mismatched alloc/free checking: a new flag

  --show-mismatched-frees=no|yes [yes] makes it possible to turn off
  such checks if necessary.

* Helgrind:

 - Improvements to error messages:

   o Race condition error message involving heap allocated blocks also
     show the thread number that allocated the raced-on block.

   o All locks referenced by an error message are now announced.
     Previously, some error messages only showed the lock addresses.

   o The message indicating where a lock was first observed now also
     describes the address/location of the lock.

 - Helgrind now understands the Ada task termination rules and
   creates a happens-before relationship between a terminated task
   and its master. This avoids some false positives and avoids a big
   memory leak when a lot of Ada tasks are created and terminated.
   The interceptions are only activated with forthcoming releases of
   gnatpro >= 7.3.0w-20140611 and gcc >= 5.0.

 - A new GDB server monitor command "info locks" giving the list of
   locks, their location, and their status.

* Callgrind:

 - callgrind_control now supports the --vgdb-prefix argument,
   which is needed if valgrind was started with this same argument.

* ==================== OTHER CHANGES ====================

* Unwinding through inlined function calls. Stack unwinding can now
  make use of Dwarf3 inlined-unwind information if it is available.
  The practical effect is that inlined calls become visible in stack
  traces. The suppression matching machinery has been adjusted
  accordingly. This is controlled by the new option
  --read-inline-info=yes|no. Currently this is enabled by default
  only on Linux and Android targets and only for the tools Memcheck,
  Helgrind and DRD.

* Valgrind can now read EXIDX unwind information on 32-bit ARM
  targets. If an object contains both CFI and EXIDX unwind
  information, Valgrind will prefer the CFI over the EXIDX. This
  facilitates unwinding through system libraries on arm-android
  targets.

* Address description logic has been improved and is now common
  between Memcheck and Helgrind, resulting in better address
  descriptions for some kinds of error messages.

* Error messages about dubious arguments (eg, to malloc or calloc) are
  output like other errors. This means that they can be suppressed

and they have a stack trace.

* The C++ demangler has been updated for better C++11 support.

* New and modified GDB server monitor features:

  - Thread local variables/storage (__thread) can now be displayed.

  - The GDB server monitor command "v.info location <address>"
    displays information about an address. The information produced
    depends on the tool and on the options given to valgrind.
    Possibly, the following are described: global variables, local
    (stack) variables, allocated or freed blocks, ...

  - The option "--vgdb-stop-at=event1,event2,..." allows the user to
    ask the GDB server to stop at the start of program execution, at
    the end of the program execution and on Valgrind internal errors.

  - A new monitor command "v.info stats" shows various Valgrind core
    and tool statistics.

  - A new monitor command "v.set hostvisibility" allows the GDB server
    to provide access to Valgrind internal host status/memory.

* A new option "--aspace-minaddr=<address>" can in some situations
  allow the use of more memory by decreasing the address above which
  Valgrind maps memory. It can also be used to solve address
  conflicts with system libraries by increasing the default value.
  See user manual for details.

* The amount of memory used by Valgrind to store debug info (unwind
  info, line number information and symbol data) has been
  significantly reduced, even though Valgrind now reads more
  information in order to support unwinding of inlined function calls.

* Dwarf3 handling with --read-var-info=yes has been improved:

  - Ada and C struct containing VLAs no longer cause a "bad DIE" error

  - Code compiled with
    -ffunction-sections -fdata-sections -Wl,--gc-sections
    no longer causes assertion failures.

* Improved checking for the --sim-hints= and --kernel-variant=
  options. Unknown strings are now detected and reported to the user
  as a usage error.

* The semantics of stack start/end boundaries in the valgrind.h
  VALGRIND_STACK_REGISTER client request has been clarified and
  documented. The convention is that start and end are respectively
  the lowest and highest addressable bytes of the stack.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

```
175819  Support for ipv6 socket reporting with --track-fds
232510  make distcheck fails
249435  Analyzing wine programs with callgrind triggers a crash
278972  support for inlined function calls in stacktraces and suppression
        == 199144
291310  FXSAVE instruction marks memory as undefined on amd64
303536  ioctl for SIOCETHTOOL (ethtool(8)) isn't wrapped
308729  vex x86->IR: unhandled instruction bytes 0xf 0x5 (syscall)
315199  vgcore file for threaded app does not show which thread crashed
315952  tun/tap ioctls are not supported
323178  Unhandled instruction: PLDW register (ARM)
323179  Unhandled instruction: PLDW immediate (ARM)
324050  Helgrind: SEGV because of unaligned stack when using movdqa
325110  Add test-cases for Power ISA 2.06 insns: divdo/divdo. and divduo/divduo.
325124  [MIPSEL] Compilation error
325477  Phase 4 support for IBM Power ISA 2.07
325538  cavium octeon mips64, valgrind reported "dumping core" [...]
325628  Phase 5 support for IBM Power ISA 2.07
325714  Empty vgcore but RLIMIT_CORE is big enough (too big)
325751  Missing the two privileged Power PC Transactional Memory Instructions
325816  Phase 6 support for IBM Power ISA 2.07
325856  Make SGCheck fail gracefully on unsupported platforms
326026  Iop names for count leading zeros/sign bits incorrectly imply [..]
326436  DRD: False positive in libstdc++ std::list::push_back
326444  Cavium MIPS Octeon Specific Load Indexed Instructions
326462  Refactor vgdb to isolate invoker stuff into separate module
326469  amd64->IR: 0x66 0xF 0x3A 0x63 0xC1 0xE (pcmpistri 0x0E)
326623  DRD: false positive conflict report in a field assignment
326724  Valgrind does not compile on OSX 1.9 Mavericks
326816  Intercept for __strncpy_sse2_unaligned missing?
326921  coregrind fails to compile m_trampoline.S with MIPS/Linux port of V
326983  Clear direction flag after tests on amd64.
327212  Do not prepend the current directory to absolute path names.
327223  Support for Cavium MIPS Octeon Atomic and Count Instructions
327238  Callgrind Assertion 'passed <= last_bb->cjmp_count' failed
327284  s390x: Fix translation of the risbg instruction
327639  vex amd64->IR pcmpestri SSE4.2 instruction is unsupported 0x34
327837  dwz compressed alternate .debug_info and .debug_str not read correctly
327916  DW_TAG_typedef may have no name
327943  s390x: add a redirection for the 'index' function
328100  XABORT not implemented
328205  Implement additional Xen hypercalls
328454  add support Backtraces with ARM unwind tables (EXIDX)
```

328455 s390x: SIGILL after emitting wrong register pair for ldxbr
328711 valgrind.1 manpage "memcheck options" section is badly generated
328878 vex amd64->IR pcmpestri SSE4.2 instruction is unsupported 0x14
329612 Incorrect handling of AT_BASE for image execution
329694 clang warns about using uninitialized variable
329956 valgrind crashes when lmw/stmw instructions are used on ppc64
330228 mmap must align to VKI_SHMLBA on mips32
330257 LLVM does not support `-mno-dynamic-no-pic` option
330319 amd64->IR: unhandled instruction bytes: 0xF 0x1 0xD5 (xend)
330459 --track-fds=yes doesn't track eventfds
330469 Add clock_adjtime syscall support
330594 Missing sysalls on PowerPC / uClibc
330622 Add test to regression suite for POWER instruction: dcbzl
330939 Support for AMD's syscall instruction on x86
== 308729
330941 Typo in PRE(poll) syscall wrapper
331057 unhandled instruction: 0xEEE01B20 (vfma.f64) (has patch)
331254 Fix expected output for memcheck/tests/dw4
331255 Fix race condition in test none/tests/coolo_sigaction
331257 Fix type of jump buffer in test none/tests/faultstatus
331305 configure uses bash specific syntax
331337 s390x WARNING: unhandled syscall: 326 (dup3)
331380 Syscall param timer_create(evp) points to uninitialised byte(s)
331476 Patch to handle ioctl 0x5422 on Linux (x86 and amd64)
331829 Unexpected ioctl opcode sign extension
331830 ppc64: WARNING: unhandled syscall: 96/97
331839 drd/tests/sem_open specifies invalid semaphore name
331847 outcome of drd/tests/thread_name is nondeterministic
332037 Valgrind cannot handle Thumb "add pc, reg"
332055 drd asserts on platforms with VG_STACK_REDZONE_SZB == 0 and
consistency checks enabled
332263 intercepts for pthread_rwlock_timedrdlock and
pthread_rwlock_timedwrlock are incorrect
332265 drd could do with post-rwlock_init and pre-rwlock_destroy
client requests
332276 Implement additional Xen hypercalls
332658 ldrd.w r1, r2, [PC, #imm] does not adjust for 32bit alignment
332765 Fix ms_print to create temporary files in a proper directory
333072 drd: Add semaphore annotations
333145 Tests for missaligned PC+#imm access for arm
333228 AAarch64 Missing instruction encoding: mrs %[reg], ctr_el0
333230 AAarch64 missing instruction encodings: dc, ic, dsb.
333248 WARNING: unhandled syscall: unix:443
333428 ldr.w pc [rD, #imm] instruction leads to assertion
333501 cachegrind: assertion: Cache set count is not a power of two.
== 336577
== 292281
333666 Recognize MPX instructions and bnd prefix.
333788 Valgrind does not support the CDROM_DISC_STATUS ioctl (has patch)
333817 Valgrind reports the memory areas written to by the SG_IO
ioctl as untouched
334049 lzcnt fails silently (x86_32)
334384 Valgrind does not have support Little Endian support for
IBM POWER PPC 64

334585 recvmmsg unhandled (+patch) (arm)
334705 sendmsg and recvmsg should guard against bogus msghdr fields.
334727 Build fails with -Werror=format-security
334788 clarify doc about --log-file initial program directory
334834 PPC64 Little Endian support, patch 2
334836 PPC64 Little Endian support, patch 3 testcase fixes
334936 patch to fix false positives on alsa SNDRV_CTL_* ioctls
335034 Unhandled ioctl: HCIGETDEVLIST
335155 vgdb, fix error print statement.
335262 arm64: movi 8bit version is not supported
335263 arm64: dmb instruction is not implemented
335441 unhandled ioctl 0x8905 (SIOCATMARK) when running wine under valgrind
335496 arm64: sbc/abc instructions are not implemented
335554 arm64: unhandled instruction: abs
335564 arm64: unhandled instruction: fcvtpu Xn, Sn
335735 arm64: unhandled instruction: cnt
335736 arm64: unhandled instruction: uaddlv
335848 arm64: unhandled instruction: {s,u}cvtf
335902 arm64: unhandled instruction: sli
335903 arm64: unhandled instruction: umull (vector)
336055 arm64: unhandled instruction: mov (element)
336062 arm64: unhandled instruction: shrn{,2}
336139 mip64: [...] valgrind hangs and spins on a single core [...]
336189 arm64: unhandled Instruction: mvn
336435 Valgrind hangs in pthread_spin_lock consuming 100% CPU
336619 valgrind --read-var-info=yes doesn't handle DW_TAG_restrict_type
336772 Make moans about unknown ioctls more informative
336957 Add a section about the Solaris/illumos port on the webpage
337094 ifunc wrapper is broken on ppc64
337285 fcntl commands F_OFD_SETLK, F_OFD_SETLKW, and F_OFD_GETLK not supported
337528 leak check heuristic for block prefixed by length as 64bit number
337740 Implement additional Xen hypercalls
337762 guest_arm64_toIR.c:4166 (dis_ARM64_load_store): Assertion `0' failed.
337766 arm64-linux: unhandled syscalls mlock (228) and mlockall (230)
337871 deprecate --db-attach
338023 Add support for all V4L2/media ioctls
338024 inlined functions are not shown if DW_AT_ranges is used
338106 Add support for 'kcmp' syscall
338115 DRD: computed conflict set differs from actual after fork
338160 implement display of thread local storage in gdbsrv
338205 configure.ac and check for -Wno-tautological-compare
338300 coredumps are missing one byte of every segment
338445 amd64 vbit-test fails with unknown opcodes used by arm64 VEX
338499 --sim-hints parsing broken due to wrong order in tokens
338615 suppress glibc 2.20 optimized strcmp implementation for ARMv7
338681 Unable to unwind through clone thread created on i386-linux
338698 race condition between gdbsrv and vgdb on startup
338703 helgrind on arm-linux gets false positives in dynamic loader
338791 alt dwz files can be relative of debug/main file
338878 on MacOS: assertion 'VG_IS_PAGE_ALIGNED(clstack_end+1)' failed
338932 build V-trunk with gcc-trunk
338974 glibc 2.20 changed size of struct sigaction sa_flags field on s390
345079 Fix build problems in VEX/useful/test_main.c
n-i-bz Fix KVM_CREATE_IRQCHIP ioctl handling

n-i-bz s390x: Fix memory corruption for multithreaded applications
n-i-bz vex arm->IR: allow PC as basereg in some LDRD cases
n-i-bz internal error in Valgrind if vgdb transmit signals when ptrace invoked
n-i-bz Fix mingw64 support in valgrind.h (dev@, 9 May 2014)
n-i-bz drd manual: Document how to C++11 programs that use class "std::thread"
n-i-bz Add command-line option --default-suppressions
n-i-bz Add support for BLKDISCARDZEROES ioctl
n-i-bz ppc32/64: fix a regression with the mtfsb0/mtfsb1 instructions
n-i-bz Add support for sys_pivot_root and sys_unshare

(3.10.0.BETA1: 2 September 2014, vex r2940, valgrind r14428)
(3.10.0.BETA2: 8 September 2014, vex r2950, valgrind r14503)
(3.10.0:   10 September 2014, vex r2950, valgrind r14514)

Release 3.9.0 (31 October 2013)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3.9.0 is a feature release with many improvements and the usual
collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux,
PPC64/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android,
X86/Android, X86/MacOSX 10.7 and AMD64/MacOSX 10.7. Support for
MacOSX 10.8 is significantly improved relative to the 3.8.0 release.

* ================== PLATFORM CHANGES =================

* Support for MIPS64 LE and BE running Linux. Valgrind has been
  tested on MIPS64 Debian Squeeze and Debian Wheezy distributions.

* Support for MIPS DSP ASE on MIPS32 platforms.

* Support for s390x Decimal Floating Point instructions on hosts that
  have the DFP facility installed.

* Support for POWER8 (Power ISA 2.07) instructions

* Support for Intel AVX2 instructions. This is available only on 64
  bit code.

* Initial support for Intel Transactional Synchronization Extensions,
  both RTM and HLE.

* Initial support for Hardware Transactional Memory on POWER.

* Improved support for MacOSX 10.8 (64-bit only). Memcheck can now
  run large GUI apps tolerably well.

* ==================== TOOL CHANGES ====================

* Memcheck:

  - Improvements in handling of vectorised code, leading to
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

significantly fewer false error reports. You need to use the flag
--partial-loads-ok=yes to get the benefits of these changes.

- Better control over the leak checker. It is now possible to specify which leak kinds (definite/indirect/possible/reachable) should be displayed, which should be regarded as errors, and which should be suppressed by a given leak suppression. This is done using the options --show-leak-kinds=kind1,kind2,.., --errors-for-leak-kinds=kind1,kind2,.. and an optional "match-leak-kinds:" line in suppression entries, respectively.

  Note that generated leak suppressions contain this new line and are therefore more specific than in previous releases. To get the same behaviour as previous releases, remove the "match-leak-kinds:" line from generated suppressions before using them.

- Reduced "possible leak" reports from the leak checker by the use of better heuristics. The available heuristics provide detection of valid interior pointers to std::stdstring, to new[] allocated arrays with elements having destructors and to interior pointers pointing to an inner part of a C++ object using multiple inheritance. They can be selected individually using the option --leak-check-heuristics=heur1,heur2,...

- Better control of stacktrace acquisition for heap-allocated blocks. Using the --keep-stacktraces option, it is possible to control independently whether a stack trace is acquired for each allocation and deallocation. This can be used to create better "use after free" errors or to decrease Valgrind's resource consumption by recording less information.

- Better reporting of leak suppression usage. The list of used suppressions (shown when the -v option is given) now shows, for each leak suppressions, how many blocks and bytes it suppressed during the last leak search.

* Helgrind:

  - False errors resulting from the use of statically initialised mutexes and condition variables (PTHREAD_MUTEX_INITIALISER, etc) have been removed.

  - False errors resulting from the use of pthread_cond_waits that timeout, have been removed.

* ==================== OTHER CHANGES ====================

* Some attempt to tune Valgrind's space requirements to the expected capabilities of the target:

  - The default size of the translation cache has been reduced from 8 sectors to 6 on Android platforms, since each sector occupies about 40MB when using Memcheck.

  - The default size of the translation cache has been increased to 16
    sectors on all other platforms, reflecting the fact that large
    applications require instrumentation and storage of huge amounts
    of code. For similar reasons, the number of memory mapped
    segments that can be tracked has been increased by a factor of 6.

  - In all cases, the maximum number of sectors in the translation
    cache can be controlled by the new flag --num-transtab-sectors.

* Changes in how debug info (line numbers, etc) is read:

  - Valgrind no longer temporarily mmaps the entire object to read
    from it. Instead, reading is done through a small fixed sized
    buffer. This avoids virtual memory usage spikes when Valgrind
    reads debuginfo from large shared objects.

  - A new experimental remote debug info server. Valgrind can read
    debug info from a different machine (typically, a build host)
    where debuginfo objects are stored. This can save a lot of time
    and hassle when running Valgrind on resource-constrained targets
    (phones, tablets) when the full debuginfo objects are stored
    somewhere else. This is enabled by the --debuginfo-server=
    option.

  - Consistency checking between main and debug objects can be
    disabled using the --allow-mismatched-debuginfo option.

* Stack unwinding by stack scanning, on ARM. Unwinding by stack
  scanning can recover stack traces in some cases when the normal
  unwind mechanisms fail. Stack scanning is best described as "a
  nasty, dangerous and misleading hack" and so is disabled by default.
  Use --unw-stack-scan-thresh and --unw-stack-scan-frames to enable
  and control it.

* Detection and merging of recursive stack frame cycles. When your
  program has recursive algorithms, this limits the memory used by
  Valgrind for recorded stack traces and avoids recording
  uninteresting repeated calls. This is controlled by the command
  line option --merge-recursive-frame and by the monitor command
  "v.set merge-recursive-frames".

* File name and line numbers for used suppressions. The list of used
  suppressions (shown when the -v option is given) now shows, for each
  used suppression, the file name and line number where the suppression
  is defined.

* New and modified GDB server monitor features:

  - valgrind.h has a new client request, VALGRIND_MONITOR_COMMAND,
    that can be used to execute gdbserver monitor commands from the
    client program.

  - A new monitor command, "v.info open_fds", that gives the list of
    open file descriptors and additional details.

  - An optional message in the "v.info n_errs_found" monitor command,
    for example "v.info n_errs_found test 1234 finished", allowing a
    comment string to be added to the process output, perhaps for the
    purpose of separating errors of different tests or test phases.

  - A new monitor command "v.info execontext" that shows information
    about the stack traces recorded by Valgrind.

  - A new monitor command "v.do expensive_sanity_check_general" to run
    some internal consistency checks.

* New flag --sigill-diagnostics to control whether a diagnostic
  message is printed when the JIT encounters an instruction it can't
  translate. The actual behavior -- delivery of SIGILL to the
  application -- is unchanged.

* The maximum amount of memory that Valgrind can use on 64 bit targets
  has been increased from 32GB to 64GB. This should make it possible
  to run applications on Memcheck that natively require up to about 35GB.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

123837 system call: 4th argument is optional, depending on cmd
135425 memcheck should tell you where Freed blocks were Mallocd
164485 VG_N_SEGNAMES and VG_N_SEGMENTS are (still) too small
207815 Adds some of the drm ioctls to syswrap-linux.c
251569 vex amd64->IR: 0xF 0x1 0xF9 0xBF 0x90 0xD0 0x3 0x0 (RDTSCP)
252955 Impossible to compile with ccache
253519 Memcheck reports auxv pointer accesses as invalid reads.
263034 Crash when loading some PPC64 binaries
269599 Increase deepest backtrace
274695 s390x: Support "compare to/from logical" instructions (z196)
275800 s390x: Autodetect cache info (part 2)
280271 Valgrind reports possible memory leaks on still-reachable std::string
284540 Memcheck shouldn't count suppressions matching still-reachable [..]
289578 Backtraces with ARM unwind tables (stack scan flags)
296311 Wrong stack traces due to -fomit-frame-pointer (x86)
304832 ppc32: build failure
305431 Use find_buildid shdr fallback for separate .debug files
305728 Add support for AVX2 instructions
305948 ppc64: code generation for ShlD64 / ShrD64 asserts
306035 s390x: Fix IR generation for LAAG and friends

306054 s390x: Condition code computation for convert-to-int/logical
306098 s390x: alternate opcode form for convert to/from fixed
306587 Fix cache line detection from auxiliary vector for PPC.
306783 Mips unhandled syscall : 4025 / 4079 / 4182
307038 DWARF2 CFI reader: unhandled DW_OP_ opcode 0x8 (DW_OP_const1u et al)
307082 HG false positive: pthread_cond_destroy: destruction of unknown CV
307101 sys_capget second argument can be NULL
307103 sys_openat: If pathname is absolute, then dirfd is ignored.
307106 amd64->IR: f0 0f c0 02 (lock xadd byte)
307113 s390x: DFP support
307141 valgrind does't work in mips-linux system
307155 filter_gdb should filter out syscall-template.S T_PSEUDO
307285 x86_amd64 feature test for avx in test suite is wrong
307290 memcheck overlap testcase needs memcpy version filter
307463 Please add "&limit=0" to the "all open bugs" link
307465 --show-possibly-lost=no should reduce the error count / exit code
307557 Leaks on Mac OS X 10.7.5 libraries at ImageLoader::recursiveInit[..]
307729 pkgconfig support broken valgrind.pc
307828 Memcheck false errors SSE optimized wcscpy, wcscmp, wcsrchr, wcschr
307955 Building valgrind 3.7.0-r4 fails in Gentoo AMD64 when using clang
308089 Unhandled syscall on ppc64: prctl
308135 PPC32 MPC8xx has 16 bytes cache size
308321 testsuite memcheck filter interferes with gdb_filter
308333 == 307106
308341 vgdb should report process exit (or fatal signal)
308427 s390 memcheck reports tsearch cjump/cmove depends on uninit
308495 Remove build dependency on installed Xen headers
308573 Internal error on 64-bit instruction executed in 32-bit mode
308626 == 308627
308627 pmovmskb validity bit propagation is imprecise
308644 vgdb command for having the info for the track-fds option
308711 give more info about aspacemgr and arenas in out_of_memory
308717 ARM: implement fixed-point VCVT.F64.[SU]32
308718 ARM implement SMLALBB family of instructions
308886 Missing support for PTRACE_SET/GETREGSET
308930 syscall name_to_handle_at (303 on amd64) not handled
309229 V-bit tester does not report number of tests generated
309323 print unrecognized instuction on MIPS
309425 Provide a --sigill-diagnostics flag to suppress illegal [..]
309427 SSE optimized stpncpy trigger uninitialised value [..] errors
309430 Self hosting ppc64 encounters a vassert error on operand type
309600 valgrind is a bit confused about 0-sized sections
309823 Generate errors for still reachable blocks
309921 PCMPISTRI validity bit propagation is imprecise
309922 none/tests/ppc64/test_dfp5 sometimes fails
310169 The Iop_CmpORD class of Iops is not supported by the vbit checker.
310424 --read-var-info does not properly describe static variables
310792 search additional path for debug symbols
310931 s390x: Message-security assist (MSA) instruction extension [..]
311100 PPC DFP implementation of the integer operands is inconsistent [..]
311318 ARM: "128-bit constant is not implemented" error message
311407 ssse3 bcopy (actually converted memcpy) causes invalid read [..]
311690 V crashes because it redirects branches inside of a redirected function
311880 x86_64: make regtest hangs at shell_valid1

311922 WARNING: unhandled syscall: 170
311933 == 251569
312171 ppc: insn selection for DFP
312571 Rounding mode call wrong for the DFP Iops [..]
312620 Change to Iop_D32toD64 [..] for s390 DFP support broke ppc [..]
312913 Dangling pointers error should also report the alloc stack trace
312980 Building on Mountain Lion generates some compiler warnings
313267 Adding MIPS64/Linux port to Valgrind
313348 == 251569
313354 == 251569
313811 Buffer overflow in assert_fail
314099 coverity pointed out error in VEX guest_ppc_toIR.c insn_suffix
314269 ppc: dead code in insn selection
314718 ARM: implement integer divide instruction (sdiv and udiv)
315345 cl-format.xml and callgrind/dump.c don't agree on using cfl= or cfi=
315441 sendmsg syscall should ignore unset msghdr msg_flags
315534 msgrcv inside a thread causes valgrind to hang (block)
315545 Assertion '(UChar*)sec->tt[tteNo].tcptr <= (UChar*)hcode' failed
315689 disInstr(thumb): unhandled instruction: 0xF852 0x0E10 (LDRT)
315738 disInstr(arm): unhandled instruction: 0xEEBE0BEE (vcvt.s32.f64)
315959 valgrind man page has bogus SGCHECK (and no BBV) OPTIONS section
316144 valgrind.1 manpage contains unknown ??? strings [..]
316145 callgrind command line options in manpage reference (unknown) [..]
316145 callgrind command line options in manpage reference [..]
316181 drd: Fixed a 4x slowdown for certain applications
316503 Valgrind does not support SSE4 "movntdqa" instruction
316535 Use of |signed int| instead of |size_t| in valgrind messages
316696 fluidanimate program of parsec 2.1 stuck
316761 syscall open_by_handle_at (304 on amd64, 342 on x86) not handled
317091 Use -Wl,-Ttext-segment when static linking if possible [..]
317186 "Impossible happens" when occurs VCVT instruction on ARM
317318 Support for Threading Building Blocks "scalable_malloc"
317444 amd64->IR: 0xC4 0x41 0x2C 0xC2 0xD2 0x8 (vcmpeq_uqps)
317461 Fix BMI assembler configure check and avx2/bmi/fma vgtest prereqs
317463 bmi testcase IR SANITY CHECK FAILURE
317506 memcheck/tests/vbit-test fails with unknown opcode after [..]
318050 libmpiwrap fails to compile with out-of-source build
318203 setsockopt handling needs to handle SOL_SOCKET/SO_ATTACH_FILTER
318643 annotate_trace_memory tests infinite loop on arm and ppc [..]
318773 amd64->IR: 0xF3 0x48 0x0F 0xBC 0xC2 0xC3 0x66 0x0F
318929 Crash with: disInstr(thumb): 0xF321 0x0001 (ssat16)
318932 Add missing PPC64 and PPC32 system call support
319235 --db-attach=yes is broken with Yama (ptrace scoping) enabled
319395 Crash with unhandled instruction on STRT (Thumb) instructions
319494 VEX Makefile-gcc standalone build update after r2702
319505 [MIPSEL] Crash: unhandled UNRAY operator.
319858 disInstr(thumb): unhandled instruction on instruction STRBT
319932 disInstr(thumb): unhandled instruction on instruction STRHT
320057 Problems when we try to mmap more than 12 memory pages on MIPS32
320063 Memory from PTRACE_GET_THREAD_AREA is reported uninitialised
320083 disInstr(thumb): unhandled instruction on instruction LDRBT
320116 bind on AF_BLUETOOTH produces warnings because of sockaddr_rc padding
320131 WARNING: unhandled syscall: 369 on ARM (prlimit64)
320211 Stack buffer overflow in ./coregrind/m_main.c with huge TMPDIR

320661 vgModuleLocal_read_elf_debug_info(): "Assertion '!di->soname'
320895 add fanotify support (patch included)
320998 vex amd64->IR pcmpestri and pcmpestrm SSE4.2 instruction
321065 Valgrind updates for Xen 4.3
321148 Unhandled instruction: PLI (Thumb 1, 2, 3)
321363 Unhandled instruction: SSAX (ARM + Thumb)
321364 Unhandled instruction: SXTAB16 (ARM + Thumb)
321466 Unhandled instruction: SHASX (ARM + Thumb)
321467 Unhandled instruction: SHSAX (ARM + Thumb)
321468 Unhandled instruction: SHSUB16 (ARM + Thumb)
321619 Unhandled instruction: SHSUB8 (ARM + Thumb)
321620 Unhandled instruction: UASX (ARM + Thumb)
321621 Unhandled instruction: USAX (ARM + Thumb)
321692 Unhandled instruction: UQADD16 (ARM + Thumb)
321693 Unhandled instruction: LDRSBT (Thumb)
321694 Unhandled instruction: UQASX (ARM + Thumb)
321696 Unhandled instruction: UQSAX (Thumb + ARM)
321697 Unhandled instruction: UHASX (ARM + Thumb)
321703 Unhandled instruction: UHSAX (ARM + Thumb)
321704 Unhandled instruction: REVSH (ARM + Thumb)
321730 Add cg_diff and cg_merge man pages
321738 Add vgdb and valgrind-listener man pages
321814 == 315545
321891 Unhandled instruction: LDRHT (Thumb)
321960 pthread_create() then alloca() causing invalid stack write errors
321969 ppc32 and ppc64 don't support [lf]setxattr
322254 Show threadname together with tid if set by application
322294 Add initial support for IBM Power ISA 2.07
322368 Assertion failure in wqthread_hijack under OS X 10.8
322563 vex mips->IR: 0x70 0x83 0xF0 0x3A
322807 VALGRIND_PRINTF_BACKTRACE writes callstack to xml and text to stderr
322851 0bXXX binary literal syntax is not standard
323035 Unhandled instruction: LDRSHT(Thumb)
323036 Unhandled instruction: SMMLS (ARM and Thumb)
323116 The memcheck/tests/ppc64/power_ISA2_05.c fails to build [..]
323175 Unhandled instruction: SMLALD (ARM + Thumb)
323177 Unhandled instruction: SMLSLD (ARM + Thumb)
323432 Calling pthread_cond_destroy() or pthread_mutex_destroy() [..]
323437 Phase 2 support for IBM Power ISA 2.07
323713 Support mmxext (integer sse) subset on i386 (athlon)
323803 Transactional memory instructions are not supported for Power
323893 SSE3 not available on amd cpus in valgrind
323905 Probable false positive from Valgrind/drd on close()
323912 valgrind.h header isn't compatible for mingw64
324047 Valgrind doesn't support [LDR,ST]{S}[B,H]T ARM instructions
324149 helgrind: When pthread_cond_timedwait returns ETIMEDOUT [..]
324181 mmap does not handle MAP_32BIT
324227 memcheck false positive leak when a thread calls exit+block [..]
324421 Support for fanotify API on ARM architecture
324514 gdbserver monitor cmd output behaviour consistency [..]
324518 ppc64: Emulation of dcbt instructions does not handle [..]
324546 none/tests/ppc32 test_isa_2_07_part2 requests -m64
324582 When access is made to freed memory, report both allocation [..]
324594 Fix overflow computation for Power ISA 2.06 insns: mulldo/mulldo.

324765 ppc64: illegal instruction when executing none/tests/ppc64/jm-misc
324816 Incorrect VEX implementation for xscvspdp/xvcvspdp for SNaN inputs
324834 Unhandled instructions in Microsoft C run-time for x86_64
324894 Phase 3 support for IBM Power ISA 2.07
326091 drd: Avoid false race reports from optimized strlen() impls
326113 valgrind libvex hwcaps error on AMD64
n-i-bz Some wrong command line options could be ignored
n-i-bz patch to allow fair-sched on android
n-i-bz report error for vgdb snapshot requested before execution
n-i-bz same as 303624 (fixed in 3.8.0), but for x86 android

(3.9.0: 31 October 2013, vex r2796, valgrind r13708)

Release 3.8.1 (19 September 2012)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3.8.1 is a bug fix release. It fixes some assertion failures in 3.8.0 that occur moderately frequently in real use cases, adds support for some missing instructions on ARM, and fixes a deadlock condition on MacOSX. If you package or deliver 3.8.0 for others to use, you might want to consider upgrading to 3.8.1 instead.

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

284004 == 301281
289584 Unhandled instruction: 0xF 0x29 0xE5 (MOVAPS)
295808 amd64->IR: 0xF3 0xF 0xBC 0xC0 (TZCNT)
298281 wcslen causes false(?) uninitialised value warnings
301281 valgrind hangs on OS X when the process calls system()
304035 disInstr(arm): unhandled instruction 0xE1023053
304867 implement MOVBE instruction in x86 mode
304980 Assertion 'lo <= hi' failed in vgModuleLocal_find_rx_mapping
305042 amd64: implement 0F 7F encoding of movq between two registers
305199 ARM: implement QDADD and QDSUB
305321 amd64->IR: 0xF 0xD 0xC (prefetchw)
305513 killed by fatal signal: SIGSEGV
305690 DRD reporting invalid semaphore when sem_trywait fails
305926 Invalid alignment checks for some AVX instructions
306297 disInstr(thumb): unhandled instruction 0xE883 0x000C
306310 3.8.0 release tarball missing some files
306612 RHEL 6 glibc-2.X default suppressions need /lib*/libc-*patterns
306664 vex amd64->IR: 0x66 0xF 0x3A 0x62 0xD1 0x46 0x66 0xF
n-i-bz shmat of a segment > 4Gb does not work
n-i-bz simulate_control_c script wrong USR1 signal number on mips
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

n-i-bz vgdb ptrace calls wrong on mips [...]
n-i-bz Fixes for more MPI false positives
n-i-bz exp-sgcheck's memcpy causes programs to segfault
n-i-bz OSX build w/ clang: asserts at startup
n-i-bz Incorrect undef'dness prop for Iop_DPBtoBCD and Iop_BCDtoDPB
n-i-bz fix a couple of union tag-vs-field mixups
n-i-bz OSX: use __NR_poll_nocancel rather than __NR_poll

The following bugs were fixed in 3.8.0 but not listed in this NEWS
file at the time:

254088 Valgrind should know about UD2 instruction
301280 == 254088
301902 == 254088
304754 NEWS blows TeX's little mind

(3.8.1: 19 September 2012, vex r2537, valgrind r12996)

Release 3.8.0 (10 August 2012)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3.8.0 is a feature release with many improvements and the usual
collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux,
PPC64/Linux, S390X/Linux, MIPS/Linux, ARM/Android, X86/Android,
X86/MacOSX 10.6/10.7 and AMD64/MacOSX 10.6/10.7. Support for recent
distros and toolchain components (glibc 2.16, gcc 4.7) has been added.
There is initial support for MacOSX 10.8, but it is not usable for
serious work at present.

* ================== PLATFORM CHANGES =================

* Support for MIPS32 platforms running Linux. Valgrind has been
  tested on MIPS32 and MIPS32r2 platforms running different Debian
  Squeeze and MeeGo distributions. Both little-endian and big-endian
  cores are supported. The tools Memcheck, Massif and Lackey have
  been tested and are known to work. See README.mips for more details.

* Preliminary support for Android running on x86.

* Preliminary (as-yet largely unusable) support for MacOSX 10.8.

* Support for Intel AVX instructions and for AES instructions. This
  support is available only for 64 bit code.

* Support for POWER Decimal Floating Point instructions.

* ==================== TOOL CHANGES ====================

* Non-libc malloc implementations are now supported. This is useful
  for tools that replace malloc (Memcheck, Massif, DRD, Helgrind).
  Using the new option --soname-synonyms, such tools can be informed
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

that the malloc implementation is either linked statically into the executable, or is present in some other shared library different from libc.so. This makes it possible to process statically linked programs, and programs using other malloc libraries, for example TCMalloc or JEMalloc.

* For tools that provide their own replacement for malloc et al, the option --redzone-size=<number> allows users to specify the size of the padding blocks (redzones) added before and after each client allocated block. Smaller redzones decrease the memory needed by Valgrind. Bigger redzones increase the chance to detect blocks overrun or underrun. Prior to this change, the redzone size was hardwired to 16 bytes in Memcheck.

* Memcheck:

  - The leak_check GDB server monitor command now can control the maximum nr of loss records to output.

  - Reduction of memory use for applications allocating many blocks and/or having many partially defined bytes.

  - Addition of GDB server monitor command 'block_list' that lists the addresses/sizes of the blocks of a leak search loss record.

  - Addition of GDB server monitor command 'who_points_at' that lists the locations pointing at a block.

  - If a redzone size > 0 is given, VALGRIND_MALLOCLIKE_BLOCK now will detect an invalid access of these redzones, by marking them noaccess. Similarly, if a redzone size is given for a memory pool, VALGRIND_MEMPOOL_ALLOC will mark the redzones no access. This still allows to find some bugs if the user has forgotten to mark the pool superblock noaccess.

  - Performance of memory leak check has been improved, especially in cases where there are many leaked blocks and/or many suppression rules used to suppress leak reports.

  - Reduced noise (false positive) level on MacOSX 10.6/10.7, due to more precise analysis, which is important for LLVM/Clang generated code. This is at the cost of somewhat reduced performance. Note there is no change to analysis precision or costs on Linux targets.

* DRD:

  - Added even more facilities that can help finding the cause of a data race, namely the command-line option --ptrace-addr and the macro DRD_STOP_TRACING_VAR(x). More information can be found in the manual.

  - Fixed a subtle bug that could cause false positive data race reports.

* ==================== OTHER CHANGES ====================

* The C++ demangler has been updated so as to work well with C++
  compiled by up to at least g++ 4.6.

* Tool developers can make replacement/wrapping more flexible thanks
  to the new option --soname-synonyms. This was reported above, but
  in fact is very general and applies to all function
  replacement/wrapping, not just to malloc-family functions.

* Round-robin scheduling of threads can be selected, using the new
  option --fair-sched= yes. Prior to this change, the pipe-based
  thread serialisation mechanism (which is still the default) could
  give very unfair scheduling. --fair-sched=yes improves
  responsiveness of interactive multithreaded applications, and
  improves repeatability of results from the thread checkers Helgrind
  and DRD.

* For tool developers: support to run Valgrind on Valgrind has been
  improved. We can now routinely Valgrind on Helgrind or Memcheck.

* gdbserver now shows the float shadow registers as integer
  rather than float values, as the shadow values are mostly
  used as bit patterns.

* Increased limit for the --num-callers command line flag to 500.

* Performance improvements for error matching when there are many
  suppression records in use.

* Improved support for DWARF4 debugging information (bug 284184).

* Initial support for DWZ compressed Dwarf debug info.

* Improved control over the IR optimiser's handling of the tradeoff
  between performance and precision of exceptions. Specifically,
  --vex-iropt-precise-memory-exns has been removed and replaced by
  --vex-iropt-register-updates, with extended functionality. This
  allows the Valgrind gdbserver to always show up to date register
  values to GDB.

* Modest performance gains through the use of translation chaining for
  JIT-generated code.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (https://bugs.kde.org/enter_bug.cgi?product=valgrind) rather
than mailing the developers (or mailing lists) directly -- bugs that
are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
  https://bugs.kde.org/show_bug.cgi?id=XXXXXX

where XXXXXX is the bug number as listed below.

197914 Building valgrind from svn now requires automake-1.10
203877 increase to 16Mb maximum allowed alignment for memalign et al
219156 Handle statically linked malloc or other malloc lib (e.g. tcmalloc)
247386 make perf does not run all performance tests
270006 Valgrind scheduler unfair
270777 Adding MIPS/Linux port to Valgrind
270796 s390x: Removed broken support for the TS insn
271438 Fix configure for proper SSE4.2 detection
273114 s390x: Support TR, TRE, TROO, TROT, TRTO, and TRTT instructions
273475 Add support for AVX instructions
274078 improved configure logic for mpicc
276993 fix mremap 'no thrash checks'
278313 Fedora 15/x64: err read debug info with --read-var-info=yes flag
281482 memcheck incorrect byte allocation count in realloc() for silly argument
282230 group allocator for small fixed size, use it for MC_Chunk/SEc vbit
283413 Fix wrong sanity check
283671 Robustize alignment computation in LibVEX_Alloc
283961 Adding support for some HCI IOCTLs
284124 parse_type_DIE: confused by: DWARF 4
284864 == 273475 (Add support for AVX instructions)
285219 Too-restrictive constraints for Thumb2 "SP plus/minus register"
285662 (MacOSX): Memcheck needs to replace memcpy/memmove
285725 == 273475 (Add support for AVX instructions)
286261 add wrapper for linux I2C_RDWR ioctl
286270 vgpreload is not friendly to 64->32 bit execs, gives ld.so warnings
286374 Running cachegrind with --branch-sim=yes on 64-bit PowerPC program fails
286384 configure fails "checking for a supported version of gcc"
286497 == 273475 (Add support for AVX instructions)
286596 == 273475 (Add support for AVX instructions)
286917 disInstr(arm): unhandled instruction: QADD (also QSUB)
287175 ARM: scalar VFP fixed-point VCVT instructions not handled
287260 Incorrect conditional jump or move depends on uninitialised value(s)
287301 vex amd64->IR: 0x66 0xF 0x38 0x41 0xC0 0xB8 0x0 0x0 (PHMINPOSUW)
287307 == 273475 (Add support for AVX instructions)
287858 VG_(strerror): unknown error
288298 (MacOSX) unhandled syscall shm_unlink
288995 == 273475 (Add support for AVX instructions)
289470 Loading of large Mach-O thin binaries fails.
289656 == 273475 (Add support for AVX instructions)
289699 vgdb connection in relay mode erroneously closed due to buffer overrun
289823 == 293754 (PCMPxSTRx not implemented for 16-bit characters)
289839 s390x: Provide support for unicode conversion instructions
289939 monitor cmd 'leak_check' with details about leaked or reachable blocks
290006 memcheck doesn't mark %xmm as initialized after "pcmpeqw %xmm %xmm"
290655 Add support for AESKEYGENASSIST instruction
290719 valgrind-3.7.0 fails with automake-1.11.2 due to"pkglibdir" usage
290974 vgdb must align pages to VKI_SHMLBA (16KB) on ARM
291253 ES register not initialised in valgrind simulation
291568 Fix 3DNOW-related crashes with baseline x86_64 CPU (w patch)
291865 s390x: Support the "Compare Double and Swap" family of instructions
292300 == 273475 (Add support for AVX instructions)
292430 unrecognized instruction in __intel_get_new_mem_ops_cpuid

292493 == 273475 (Add support for AVX instructions)
292626 Missing fcntl F_SETOWN_EX and F_GETOWN_EX support
292627 Missing support for some SCSI ioctls
292628 none/tests/x86/bug125959-x86.c triggers undefined behavior
292841 == 273475 (Add support for AVX instructions)
292993 implement the getcpu syscall on amd64-linux
292995 Implement the “cross memory attach” syscalls introduced in Linux 3.2
293088 Add some VEX sanity checks for ppc64 unhandled instructions
293751 == 290655 (Add support for AESKEYGENASSIST instruction)
293754 PCMPxSTRx not implemented for 16-bit characters
293755 == 293754 (No tests for PCMPxSTRx on 16-bit characters)
293808 CLFLUSH not supported by latest VEX for amd64
294047 valgrind does not correctly emulate prlimit64(..., RLIMIT_NOFILE, ...)
294048 MPSADBW instruction not implemented
294055 regtest none/tests/shell fails when locale is not set to C
294185 INT 0x44 (and others) not supported on x86 guest, but used by Jikes RVM
294190 --vgdb-error=xxx can be out of sync with errors shown to the user
294191 amd64: fnsave/frstor and 0x66 size prefixes on FP instructions
294260 disInstr_AMD64: disInstr miscalculated next %rip
294523 --partial-loads-ok=yes causes false negatives
294617 vex amd64->IR: 0x66 0xF 0x3A 0xDF 0xD1 0x1 0xE8 0x6A
294736 vex amd64->IR: 0x48 0xF 0xD7 0xD6 0x48 0x83
294812 patch allowing to run (on x86 at least) helgrind/drd on tool.
295089 can not annotate source for both helgrind and drd
295221 POWER Processor decimal floating point instruction support missing
295427 building for i386 with clang on darwin11 requires "-new_linker linker"
295428 coregrind/m_main.c has incorrect x86 assembly for darwin
295590 Helgrind: Assertion 'cvi->nWaiters > 0' failed
295617 ARM - Add some missing syscalls
295799 Missing \n with get_vbits in gdbserver when line is % 80 [...]
296229 Linux user input device ioctls missing wrappers
296318 ELF Debug info improvements (more than one rx/rw mapping)
296422 Add translation chaining support
296457 vex amd64->IR: 0x66 0xF 0x3A 0xDF 0xD1 0x1 0xE8 0x6A (dup of AES)
296792 valgrind 3.7.0: add SIOCSHWTSTAMP (0x89B0) ioctl wrapper
296983 Fix build issues on x86_64/ppc64 without 32-bit toolchains
297078 gdbserver signal handling problems [..]
297147 drd false positives on newly allocated memory
297329 disallow decoding of IBM Power DFP insns on some machines
297497 POWER Processor decimal floating point instruction support missing
297701 Another alias for strncasecmp_l in libc-2.13.so
297911 'invalid write' not reported when using APIs for custom mem allocators.
297976 s390x: revisit EX implementation
297991 Valgrind interferes with mmap()+ftell()
297992 Support systems missing WIFCONTINUED (e.g. pre-2.6.10 Linux)
297993 Fix compilation of valgrind with gcc -g3.
298080 POWER Processor DFP support missing, part 3
298227 == 273475 (Add support for AVX instructions)
298335 == 273475 (Add support for AVX instructions)
298354 Unhandled ARM Thumb instruction 0xEB0D 0x0585 (streq)
298394 s390x: Don't bail out on an unknown machine model. [..]
298421 accept4() syscall (366) support is missing for ARM
298718 vex amd64->IR: 0xF 0xB1 0xCB 0x9C 0x8F 0x45
298732 valgrind installation problem in ubuntu with kernel version 3.x

298862 POWER Processor DFP instruction support missing, part 4
298864 DWARF reader mis-parses DW_FORM_ref_addr
298943 massif asserts with --pages-as-heap=yes when brk is changing [..]
299053 Support DWARF4 DW_AT_high_pc constant form
299104 == 273475 (Add support for AVX instructions)
299316 Helgrind: hg_main.c:628 (map_threads_lookup): Assertion 'thr' failed.
299629 dup3() syscall (358) support is missing for ARM
299694 POWER Processor DFP instruction support missing, part 5
299756 Ignore --free-fill for MEMPOOL_FREE and FREELIKE client requests
299803 == 273475 (Add support for AVX instructions)
299804 == 273475 (Add support for AVX instructions)
299805 == 273475 (Add support for AVX instructions)
300140 ARM - Missing (T1) SMMUL
300195 == 296318 (ELF Debug info improvements (more than one rx/rw mapping))
300389 Assertion `are_valid_hwcaps(VexArchAMD64, [..])' failed.
300414 FCOM and FCOMP unimplemented for amd64 guest
301204 infinite loop in canonicaliseSymtab with ifunc symbol
301229 == 203877 (increase to 16Mb maximum allowed alignment for memalign etc)
301265 add x86 support to Android build
301984 configure script doesn't detect certain versions of clang
302205 Fix compiler warnings for POWER VEX code and POWER test cases
302287 Unhandled movbe instruction on Atom processors
302370 PPC: fnmadd, fnmsub, fnmadds, fnmsubs insns always negate the result
302536 Fix for the POWER Valgrind regression test: memcheck-ISA2.0.
302578 Unrecognized isntruction 0xc5 0x32 0xc2 0xca 0x09 vcmpngess
302656 == 273475 (Add support for AVX instructions)
302709 valgrind for ARM needs extra tls support for android emulator [..]
302827 add wrapper for CDROM_GET_CAPABILITY
302901 Valgrind crashes with dwz optimized debuginfo
302918 Enable testing of the vmaddfp and vnsubfp instructions in the testsuite
303116 Add support for the POWER instruction popcntb
303127 Power test suite fixes for frsqrte, vrefp, and vrsqrtefp instructions.
303250 Assertion `instrs_in->arr_used <= 10000' failed w/ OpenSSL code
303466 == 273475 (Add support for AVX instructions)
303624 segmentation fault on Android 4.1 (e.g. on Galaxy Nexus OMAP)
303963 strstr() function produces wrong results under valgrind callgrind
304054 CALL_FN_xx macros need to enforce stack alignment
304561 tee system call not supported
715750 (MacOSX): Incorrect invalid-address errors near 0xFFFFxxxx (mozbug#)
n-i-bz Add missing gdbserver xml files for shadow registers for ppc32
n-i-bz Bypass gcc4.4/4.5 code gen bugs causing out of memory or asserts
n-i-bz Fix assert in gdbserver for watchpoints watching the same address
n-i-bz Fix false positive in sys_clone on amd64 when optional args [..]
n-i-bz s390x: Shadow registers can now be examined using vgdb

(3.8.0-TEST3: 9 August 2012, vex r2465, valgrind r12865)
(3.8.0: 10 August 2012, vex r2465, valgrind r12866)

Release 3.7.0 (5 November 2011)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3.7.0 is a feature release with many significant improvements and the
usual collection of bug fixes.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This release supports X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, ARM/Android, X86/Darwin and AMD64/Darwin. Support for recent distros and toolchain components (glibc 2.14, gcc 4.6, MacOSX 10.7) has been added.

* ================== PLATFORM CHANGES =================

* Support for IBM z/Architecture (s390x) running Linux. Valgrind can analyse 64-bit programs running on z/Architecture. Most user space instructions up to and including z10 are supported. Valgrind has been tested extensively on z9, z10, and z196 machines running SLES 10/11, RedHat 5/6m, and Fedora. The Memcheck and Massif tools are known to work well. Callgrind, Helgrind, and DRD work reasonably well on z9 and later models. See README.s390 for more details.

* Preliminary support for MacOSX 10.7 and XCode 4. Both 32- and 64-bit processes are supported. Some complex threaded applications (Firefox) are observed to hang when run as 32 bit applications, whereas 64-bit versions run OK. The cause is unknown. Memcheck will likely report some false errors. In general, expect some rough spots. This release also supports MacOSX 10.6, but drops support for 10.5.

* Preliminary support for Android (on ARM). Valgrind can now run large applications (eg, Firefox) on (eg) a Samsung Nexus S. See README.android for more details, plus instructions on how to get started.

* Support for the IBM Power ISA 2.06 (Power7 instructions)

* General correctness and performance improvements for ARM/Linux, and, by extension, ARM/Android.

* Further solidification of support for SSE 4.2 in 64-bit mode. AVX instruction set support is under development but is not available in this release.

* Support for AIX5 has been removed.

* ==================== TOOL CHANGES ====================

* Memcheck: some incremental changes:

  - reduction of memory use in some circumstances

  - improved handling of freed memory, which in some circumstances can cause detection of use-after-free that would previously have been missed

  - fix of a longstanding bug that could cause false negatives (missed errors) in programs doing vector saturated narrowing instructions.

* Helgrind: performance improvements and major memory use reductions,

particularly for large, long running applications which perform many synchronisation (lock, unlock, etc) events. Plus many smaller changes:

- display of locksets for both threads involved in a race

- general improvements in formatting/clarity of error messages

- addition of facilities and documentation regarding annotation of thread safe reference counted C++ classes

- new flag --check-stack-refs=no|yes [yes], to disable race checking on thread stacks (a performance hack)

- new flag --free-is-write=no|yes [no], to enable detection of races where one thread accesses heap memory but another one frees it, without any coordinating synchronisation event

* DRD: enabled XML output; added support for delayed thread deletion in order to detect races that occur close to the end of a thread (--join-list-vol); fixed a memory leak triggered by repeated client memory allocatation and deallocation; improved Darwin support.

* exp-ptrcheck: this tool has been renamed to exp-sgcheck

* exp-sgcheck: this tool has been reduced in scope so as to improve performance and remove checking that Memcheck does better. Specifically, the ability to check for overruns for stack and global arrays is unchanged, but the ability to check for overruns of heap blocks has been removed. The tool has accordingly been renamed to exp-sgcheck ("Stack and Global Array Checking").

* ==================== OTHER CHANGES ====================

* GDB server: Valgrind now has an embedded GDB server. That means it is possible to control a Valgrind run from GDB, doing all the usual things that GDB can do (single stepping, breakpoints, examining data, etc). Tool-specific functionality is also available. For example, it is possible to query the definedness state of variables or memory from within GDB when running Memcheck; arbitrarily large memory watchpoints are supported, etc. To use the GDB server, start Valgrind with the flag --vgdb-error=0 and follow the on-screen instructions.

* Improved support for unfriendly self-modifying code: a new option --smc-check=all-non-file is available. This adds the relevant consistency checks only to code that originates in non-file-backed mappings. In effect this confines the consistency checking only to code that is or might be JIT generated, and avoids checks on code that must have been compiled ahead of time. This significantly improves performance on applications that generate code at run time.

* It is now possible to build a working Valgrind using Clang-2.9 on Linux.

* new client requests VALGRIND_{DISABLE,ENABLE}_ERROR_REPORTING.
  These enable and disable error reporting on a per-thread, and
  nestable, basis. This is useful for hiding errors in particularly
  troublesome pieces of code. The MPI wrapper library (libmpiwrap.c)
  now uses this facility.

* Added the --mod-funcname option to cg_diff.

* ==================== FIXED BUGS ====================

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (http://bugs.kde.org/enter_valgrind_bug.cgi) rather than
mailing the developers (or mailing lists) directly -- bugs that are
not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

```
 79311  malloc silly arg warning does not give stack trace
210935  port valgrind.h (not valgrind) to win32 to support client requests
214223  valgrind SIGSEGV on startup gcc 4.4.1 ppc32 (G4) Ubuntu 9.10
243404  Port to zSeries
243935  Helgrind: incorrect handling of ANNOTATE_HAPPENS_BEFORE()/AFTER()
247223  non-x86: Suppress warning: 'regparm' attribute directive ignored
250101  huge "free" memory usage due to m_mallocfree.c fragmentation
253206  Some fixes for the faultstatus testcase
255223  capget testcase fails when running as root
256703  xlc_dbl_u32.c testcase broken
256726  Helgrind tests have broken inline asm
259977  == 214223 (Valgrind segfaults doing __builtin_longjmp)
264800  testcase compile failure on zseries
265762  make public VEX headers compilable by G++ 3.x
265771  assertion in jumps.c (r11523) fails with glibc-2.3
266753  configure script does not give the user the option to not use QtCore
266931  gen_insn_test.pl is broken
266961  ld-linux.so.2 i?86-linux strlen issues
266990  setns instruction causes false positive
267020  Make directory for temporary files configurable at run-time.
267342  == 267997 (segmentation fault on Mac OS 10.6)
267383  Assertion 'vgPlain_strlen(dir) + vgPlain_strlen(file) + 1 < 256' failed
267413  Assertion 'DRD_(g_threadinfo)[tid].synchr_nesting >= 1' failed.
267488  regtest: darwin support for 64-bit build
267552  SIGSEGV (misaligned_stack_error) with DRD, but not with other tools
267630  Add support for IBM Power ISA 2.06 -- stage 1
267769  == 267997 (Darwin: memcheck triggers segmentation fault)
267819  Add client request for informing the core about reallocation
267925  laog data structure quadratic for a single sequence of lock
267968  drd: (vgDrd_thread_set_joinable): Assertion '0 <= (int)tid ..' failed
267997  MacOSX: 64-bit V segfaults on launch when built with Xcode 4.0.1
268513  missed optimizations in fold_Expr
```

268619 s390x: fpr - gpr transfer facility
268620 s390x: reconsider "long displacement" requirement
268621 s390x: improve IR generation for XC
268715 s390x: FLOGR is not universally available
268792 == 267997 (valgrind seg faults on startup when compiled with Xcode 4)
268930 s390x: MHY is not universally available
269078 arm->IR: unhandled instruction SUB (SP minus immediate/register)
269079 Support ptrace system call on ARM
269144 missing "Bad option" error message
269209 conditional load and store facility (z196)
269354 Shift by zero on x86 can incorrectly clobber CC_NDEP
269641 == 267997 (valgrind segfaults immediately (segmentation fault))
269736 s390x: minor code generation tweaks
269778 == 272986 (valgrind.h: swap roles of VALGRIND_DO_CLIENT_REQUEST() ..)
269863 s390x: remove unused function parameters
269864 s390x: tweak s390_emit_load_cc
269884 == 250101 (overhead for huge blocks exhausts space too soon)
270082 s390x: Make sure to point the PSW address to the next address on SIGILL
270115 s390x: rewrite some testcases
270309 == 267997 (valgrind crash on startup)
270320 add support for Linux FIOQSIZE ioctl() call
270326 segfault while trying to sanitize the environment passed to execle
270794 IBM POWER7 support patch causes regression in none/tests
270851 IBM POWER7 fcfidus instruction causes memcheck to fail
270856 IBM POWER7 xsnmaddadp instruction causes memcheck to fail on 32bit app
270925 hyper-optimized strspn() in /lib64/libc-2.13.so needs fix
270959 s390x: invalid use of R0 as base register
271042 VSX configure check fails when it should not
271043 Valgrind build fails with assembler error on ppc64 with binutils 2.21
271259 s390x: fix code confusion
271337 == 267997 (Valgrind segfaults on MacOS X)
271385 s390x: Implement Ist_MBE
271501 s390x: misc cleanups
271504 s390x: promote likely and unlikely
271579 ppc: using wrong enum type
271615 unhandled instruction "popcnt" (arch=amd10h)
271730 Fix bug when checking ioctls: duplicate check
271776 s390x: provide STFLE instruction support
271779 s390x: provide clock instructions like STCK
271799 Darwin: ioctls without an arg report a memory error
271820 arm: fix type confusion
271917 pthread_cond_timedwait failure leads to not-locked false positive
272067 s390x: fix DISP20 macro
272615 A typo in debug output in mc_leakcheck.c
272661 callgrind_annotate chokes when run from paths containing regex chars
272893 amd64->IR: 0x66 0xF 0x38 0x2B 0xC1 0x66 0xF 0x7F == (closed as dup)
272955 Unhandled syscall error for pwrite64 on ppc64 arch
272967 make documentation build-system more robust
272986 Fix gcc-4.6 warnings with valgrind.h
273318 amd64->IR: 0x66 0xF 0x3A 0x61 0xC1 0x38 (missing PCMPxSTRx case)
273318 unhandled PCMPxSTRx case: vex amd64->IR: 0x66 0xF 0x3A 0x61 0xC1 0x38
273431 valgrind segfaults in evalCfiExpr (debuginfo.c:2039)
273465 Callgrind: jumps.c:164 (new_jcc): Assertion '(0 <= jmp) && ...'
273536 Build error: multiple definition of `vgDrd_pthread_cond_initializer'

273640 ppc64-linux: unhandled syscalls setresuid(164) and setresgid(169)
273729 == 283000 (Illegal opcode for SSE2 "roundsd" instruction)
273778 exp-ptrcheck: unhandled sysno == 259
274089 exp-ptrcheck: unhandled sysno == 208
274378 s390x: Various dispatcher tweaks
274447 WARNING: unhandled syscall: 340
274776 amd64->IR: 0x66 0xF 0x38 0x2B 0xC5 0x66
274784 == 267997 (valgrind ls -l results in Segmentation Fault)
274926 valgrind does not build against linux-3
275148 configure FAIL with glibc-2.14
275151 Fedora 15 / glibc-2.14 'make regtest' FAIL
275168 Make Valgrind work for MacOSX 10.7 Lion
275212 == 275284 (lots of false positives from __memcpy_ssse3_back et al)
275278 valgrind does not build on Linux kernel 3.0.* due to silly
275284 Valgrind memcpy/memmove redirection stopped working in glibc 2.14/x86_64
275308 Fix implementation for ppc64 fres instruc
275339 s390x: fix testcase compile warnings
275517 s390x: Provide support for CKSM instruction
275710 s390x: get rid of redundant address mode calculation
275815 == 247894 (Valgrind doesn't know about Linux readahead(2) syscall)
275852 == 250101 (valgrind uses all swap space and is killed)
276784 Add support for IBM Power ISA 2.06 -- stage 3
276987 gdbsrv: fix tests following recent commits
277045 Valgrind crashes with unhandled DW_OP_ opcode 0x2a
277199 The test_isa_2_06_part1.c in none/tests/ppc64 should be a symlink
277471 Unhandled syscall: 340
277610 valgrind crashes in VG_(lseek)(core_fd, phdrs[idx].p_offset, ...)
277653 ARM: support Thumb2 PLD instruction
277663 ARM: NEON float VMUL by scalar incorrect
277689 ARM: tests for VSTn with register post-index are broken
277694 ARM: BLX LR instruction broken in ARM mode
277780 ARM: VMOV.F32 (immediate) instruction is broken
278057 fuse filesystem syscall deadlocks
278078 Unimplemented syscall 280 on ppc32
278349 F_GETPIPE_SZ and F_SETPIPE_SZ Linux fcntl commands
278454 VALGRIND_STACK_DEREGISTER has wrong output type
278502 == 275284 (Valgrind confuses memcpy() and memmove())
278892 gdbsrv: factorize gdb version handling, fix doc and typos
279027 Support for MVCL and CLCL instruction
279027 s390x: Provide support for CLCL and MVCL instructions
279062 Remove a redundant check in the insn selector for ppc.
279071 JDK creates PTEST with redundant REX.W prefix
279212 gdbsrv: add monitor cmd v.info scheduler.
279378 exp-ptrcheck: the 'impossible' happened on mkfifo call
279698 memcheck discards valid-bits for packuswb
279795 memcheck reports uninitialised values for mincore on amd64
279994 Add support for IBM Power ISA 2.06 -- stage 3
280083 mempolicy syscall check errors
280290 vex amd64->IR: 0x66 0xF 0x38 0x28 0xC1 0x66 0xF 0x6F
280710 s390x: config files for nightly builds
280757 /tmp dir still used by valgrind even if TMPDIR is specified
280965 Valgrind breaks fcntl locks when program does mmap
281138 WARNING: unhandled syscall: 340
281241 == 275168 (valgrind useless on Macos 10.7.1 Lion)

281304 == 275168 (Darwin: dyld "cannot load inserted library")
281305 == 275168 (unhandled syscall: unix:357 on Darwin 11.1)
281468 s390x: handle do_clone and gcc clones in call traces
281488 ARM: VFP register corruption
281828 == 275284 (false memmove warning: "Source and destination overlap")
281883 s390x: Fix system call wrapper for "clone".
282105 generalise 'reclaimSuperBlock' to also reclaim splittable superblock
282112 Unhandled instruction bytes: 0xDE 0xD9 0x9B 0xDF (fcompp)
282238 SLES10: make check fails
282979 strcasestr needs replacement with recent(>=2.12) glibc
283000 vex amd64->IR: 0x66 0xF 0x3A 0xA 0xC0 0x9 0xF3 0xF
283243 Regression in ppc64 memcheck tests
283325 == 267997 (Darwin: V segfaults on startup when built with Xcode 4.0)
283427 re-connect epoll_pwait syscall on ARM linux
283600 gdbsrv: android: port vgdb.c
283709 none/tests/faultstatus needs to account for page size
284305 filter_gdb needs enhancement to work on ppc64
284384 clang 3.1 -Wunused-value warnings in valgrind.h, memcheck.h
284472 Thumb2 ROR.W encoding T2 not implemented
284621 XML-escape process command line in XML output
n-i-bz cachegrind/callgrind: handle CPUID information for Core iX Intel CPUs
that have non-power-of-2 sizes (also AMDs)
n-i-bz don't be spooked by libraries mashed by elfhack
n-i-bz don't be spooked by libxul.so linked with gold
n-i-bz improved checking for VALGRIND_CHECK_MEM_IS_DEFINED

(3.7.0-TEST1: 27 October 2011, vex r2228, valgrind r12245)
(3.7.0.RC1: 1 November 2011, vex r2231, valgrind r12257)
(3.7.0: 5 November 2011, vex r2231, valgrind r12258)

Release 3.6.1 (16 February 2011)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3.6.1 is a bug fix release. It adds support for some SSE4
instructions that were omitted in 3.6.0 due to lack of time. Initial
support for glibc-2.13 has been added. A number of bugs causing
crashing or assertion failures have been fixed.

The following bugs have been fixed or resolved. Note that "n-i-bz"
stands for "not in bugzilla" -- that is, a bug that was reported to us
but never got a bugzilla entry. We encourage you to file bugs in
bugzilla (http://bugs.kde.org/enter_valgrind_bug.cgi) rather than
mailing the developers (or mailing lists) directly -- bugs that are
not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit
https://bugs.kde.org/show_bug.cgi?id=XXXXXX
where XXXXXX is the bug number as listed below.

188572 Valgrind on Mac should suppress setenv() mem leak
194402 vex amd64->IR: 0x48 0xF 0xAE 0x4 (proper FX{SAVE,RSTOR} support)
210481 vex amd64->IR: Assertion `sz == 2 || sz == 4' failed (REX.W POPQ)
246152 callgrind internal error after pthread_cancel on 32 Bit Linux
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

250038 ppc64: Altivec LVSR and LVSL instructions fail their regtest
254420 memory pool tracking broken
254957 Test code failing to compile due to changes in memcheck.h
255009 helgrind/drd: crash on chmod with invalid parameter
255130 readdwarf3.c parse_type_DIE confused by GNAT Ada types
255355 helgrind/drd: crash on threaded programs doing fork
255358 == 255355
255418 (SSE4.x) rint call compiled with ICC
255822 --gen-suppressions can create invalid files: "too many callers [...]"
255888 closing valgrindoutput tag outputted to log-stream on error
255963 (SSE4.x) vex amd64->IR: 0x66 0xF 0x3A 0x9 0xDB 0x0 (ROUNDPD)
255966 Slowness when using mempool annotations
256387 vex x86->IR: 0xD4 0xA 0x2 0x7 (AAD and AAM)
256600 super-optimized strcasecmp() false positive
256669 vex amd64->IR: Unhandled LOOPNEL insn on amd64
256968 (SSE4.x) vex amd64->IR: 0x66 0xF 0x38 0x10 0xD3 0x66 (BLENDVPx)
257011 (SSE4.x) vex amd64->IR: 0x66 0xF 0x3A 0xE 0xFD 0xA0 (PBLENDW)
257063 (SSE4.x) vex amd64->IR: 0x66 0xF 0x3A 0x8 0xC0 0x0 (ROUNDPS)
257276 Missing case in memcheck --track-origins=yes
258870 (SSE4.x) Add support for EXTRACTPS SSE 4.1 instruction
261966 (SSE4.x) support for CRC32B and CRC32Q is lacking (also CRC32{W,L})
262985 VEX regression in valgrind 3.6.0 in handling PowerPC VMX
262995 (SSE4.x) crash when trying to valgrind gcc-snapshot (PCMPxSTRx $0)
263099 callgrind_annotate counts Ir improperly [...]
263877 undefined coprocessor instruction on ARMv7
265964 configure FAIL with glibc-2.13
n-i-bz Fix compile error w/ icc-12.x in guest_arm_toIR.c
n-i-bz Docs: fix bogus descriptions for VALGRIND_CREATE_BLOCK et al
n-i-bz Massif: don't assert on shmat() with --pages-as-heap=yes
n-i-bz Bug fixes and major speedups for the exp-DHAT space profiler
n-i-bz DRD: disable --free-is-write due to implementation difficulties

(3.6.1: 16 February 2011, vex r2103, valgrind r11561).

4. README

Release notes for Valgrind
~~~~~~~~~~~~~~~~~~~~~~~~~~~
If you are building a binary package of Valgrind for distribution,
please read README_PACKAGERS. It contains some important information.

If you are developing Valgrind, please read README_DEVELOPERS. It contains
some useful information.

For instructions on how to build/install, see the end of this file.

If you have problems, consult the FAQ to see if there are workarounds.

Executive Summary
~~~~~~~~~~~~~~~~~
Valgrind is a framework for building dynamic analysis tools. There are
Valgrind tools that can automatically detect many memory management
and threading bugs, and profile your programs in detail. You can also
use Valgrind to build new tools.

The Valgrind distribution currently includes seven production-quality
tools: a memory error detector, two thread error detectors, a cache
and branch-prediction profiler, a call-graph generating cache and
branch-prediction profiler, and two heap profilers. It also includes
one experimental tool: a SimPoint basic block vector generator.

Valgrind is closely tied to details of the CPU, operating system and to
a lesser extent, compiler and basic C libraries. This makes it difficult
to make it portable. Nonetheless, it is available for the following
platforms:

- X86/Linux
- AMD64/Linux
- PPC32/Linux
- PPC64/Linux
- ARM/Linux
- ARM64/Linux
- X86/macOS
- AMD64/macOS
- S390X/Linux
- MIPS32/Linux
- MIPS64/Linux
- nanoMIPS/Linux
- X86/Solaris
- AMD64/Solaris
- X86/FreeBSD
- AMD64/FreeBSD
- ARM64/FreeBSD

Note that AMD64 is just another name for x86_64, and Valgrind runs fine
on Intel processors. Also note that the core of macOS is called
"Darwin" and this name is used sometimes.

Valgrind is licensed under the GNU General Public License, version 2.
Read the file COPYING in the source distribution for details.

However: if you contribute code, you need to make it available as GPL
version 2 or later, and not 2-only.

Documentation
~~~~~~~~~~~~~
A comprehensive user guide is supplied. Point your browser at
$PREFIX/share/doc/valgrind/manual.html, where $PREFIX is whatever you
specified with --prefix= when building.

Building and installing it
~~~~~~~~~~~~~~~~~~~~~~~~~~
To install from the GIT repository:

0. Clone the code from GIT:
   git clone https://sourceware.org/git/valgrind.git
   There are further instructions at
   http://www.valgrind.org/downloads/repository.html.

1. cd into the source directory.

2. Run ./autogen.sh to setup the environment (you need the standard
   autoconf tools to do so).

3. Continue with the following instructions...

To install from a tar.bz2 distribution:

4. Run ./configure, with some options if you wish. The only interesting
   one is the usual --prefix=/where/you/want/it/installed.

5. Run "make".

6. Run "make install", possibly as root if the destination permissions
   require that.

7. See if it works. Try "valgrind ls -l". Either this works, or it
   bombs out with some complaint. In that case, please let us know
   (see http://valgrind.org/support/bug_reports.html).

Important! Do not move the valgrind installation into a place
different from that specified by --prefix at build time. This will
cause things to break in subtle ways, mostly when Valgrind handles
fork/exec calls.

The Valgrind Developers

5. README_MISSING_SYSCALL_OR_IOCTL

```
Dealing with missing system call or ioctl wrappers in Valgrind
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
You're probably reading this because Valgrind bombed out whilst
running your program, and advised you to read this file.  The good
news is that, in general, it's easy to write the missing syscall or
ioctl wrappers you need, so that you can continue your debugging.  If
you send the resulting patches to me, then you'll be doing a favour to
all future Valgrind users too.

Note that an "ioctl" is just a special kind of system call, really; so
there's not a lot of need to distinguish them (at least conceptually)
in the discussion that follows.

All this machinery is in coregrind/m_syswrap.


What are syscall/ioctl wrappers?  What do they do?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Valgrind does what it does, in part, by keeping track of everything your
program does.  When a system call happens, for example a request to read
part of a file, control passes to the Linux kernel, which fulfils the
request, and returns control to your program.  The problem is that the
kernel will often change the status of some part of your program's memory
as a result, and tools (instrumentation plug-ins) may need to know about
this.

Syscall and ioctl wrappers have two jobs:

1. Tell a tool what's about to happen, before the syscall takes place.  A
   tool could perform checks beforehand, eg. if memory about to be written
   is actually writable.  This part is useful, but not strictly
   essential.

2. Tell a tool what just happened, after a syscall takes place.  This is
   so it can update its view of the program's state, eg. that memory has
   just been written to.  This step is essential.

The "happenings" mostly involve reading/writing of memory.

So, let's look at an example of a wrapper for a system call which
should be familiar to many Unix programmers.


The syscall wrapper for time()
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The wrapper for the time system call looks like this:

  PRE(sys_time)
```

```
 {
   /* time_t time(time_t *t); */
   PRINT("sys_time ( %p )",ARG1);
   PRE_REG_READ1(long, "time", int *, t);
   if (ARG1 != 0) {
     PRE_MEM_WRITE( "time(t)", ARG1, sizeof(vki_time_t) );
   }
 }

 POST(sys_time)
 {
   if (ARG1 != 0) {
     POST_MEM_WRITE( ARG1, sizeof(vki_time_t) );
   }
 }
```

The first thing we do happens before the syscall occurs, in the PRE() function.
The PRE() function typically starts with invoking to the PRINT() macro. This
PRINT() macro implements support for the --trace-syscalls command line option.
Next, the tool is told the return type of the syscall, that the syscall has
one argument, the type of the syscall argument and that the argument is being
read from a register:

```
   PRE_REG_READ1(long, "time", int *, t);
```

Next, if a non-NULL buffer is passed in as the argument, tell the tool that the
buffer is about to be written to:

```
   if (ARG1 != 0) {
     PRE_MEM_WRITE( "time", ARG1, sizeof(vki_time_t) );
   }
```

Finally, the really important bit, after the syscall occurs, in the POST()
function: if, and only if, the system call was successful, tell the tool that
the memory was written:

```
   if (ARG1 != 0) {
     POST_MEM_WRITE( ARG1, sizeof(vki_time_t) );
   }
```

The POST() function won't be called if the syscall failed, so you
don't need to worry about checking that in the POST() function.
(Note: this is sometimes a bug; some syscalls do return results when
they "fail" - for example, nanosleep returns the amount of unslept
time if interrupted. TODO: add another per-syscall flag for this
case.)

Note that we use the type 'vki_time_t'. This is a copy of the kernel
type, with 'vki_' prefixed. Our copies of such types are kept in the
appropriate vki*.h file(s). We don't include kernel headers or glibc headers
directly.

Writing your own syscall wrappers (see below for ioctl wrappers)

~~~~~~

If Valgrind tells you that system call NNN is unimplemented, do the following:

1. Find out the name of the system call:

   grep NNN /usr/include/asm/unistd*.h

   This should tell you something like __NR_mysyscallname.
   Copy this entry to include/vki/vki-scnums-$(VG_PLATFORM).h.

   If you can't find the system call in /usr/include, try looking in the strace source code (https://github.com/strace/strace). Some syscalls/ioctls are not defined explicitly, but strace may have already figured it out.

2. Do 'man 2 mysyscallname' to get some idea of what the syscall does. Note that the actual kernel interface can differ from this, so you might also want to check a version of the Linux kernel source.

   NOTE: any syscall which has something to do with signals or threads is probably "special", and needs more careful handling. Post something to valgrind-developers if you aren't sure.

3. Add a case to the already-huge collection of wrappers in the coregrind/m_syswrap/syswrap-*.c files.
   For each in-memory parameter which is read or written by the syscall, do one of

   PRE_MEM_READ(...)
   PRE_MEM_RASCIIZ(...)
   PRE_MEM_WRITE(...)

   for that parameter. Then do the syscall. Then, if the syscall succeeds, issue suitable POST_MEM_WRITE(...) calls.
   (There's no need for POST_MEM_READ calls.)

   Also, add it to the syscall_table[] array; use one of GENX_, GENXY LINX_, LINXY, PLAX_, PLAXY.
   GEN* for generic syscalls (in syswrap-generic.c), LIN* for linux specific ones (in syswrap-linux.c) and PLA* for the platform dependent ones (in syswrap-$(PLATFORM)-linux.c).
   The *XY variant if it requires a PRE() and POST() function, and the *X_ variant if it only requires a PRE() function.

   If you find this difficult, read the wrappers for other syscalls for ideas. A good tip is to look for the wrapper for a syscall which has a similar behaviour to yours, and use it as a starting point.

   If you need structure definitions and/or constants for your syscall,
~~~~~~

copy them from the kernel headers into include/vki.h and co., with the appropriate vki_*/VKI_* name mangling. Don't #include any kernel headers. And certainly don't #include any glibc headers.

Test it.

Note that a common error is to call POST_MEM_WRITE(...) with 0 (NULL) as the first (address) argument. This usually means your logic is slightly inadequate. It's a sufficiently common bug that there's a built-in check for it, and you'll get a "probably sanity check failure" for the syscall wrapper you just made, if this is the case.

4. Once happy, send us the patch. Pretty please.

Writing your own ioctl wrappers

Is pretty much the same as writing syscall wrappers, except that all the action happens within PRE(ioctl) and POST(ioctl).

There's a default case, sometimes it isn't correct and you have to write a more specific case to get the right behaviour.

As above, please create a bug report and attach the patch as described on http://www.valgrind.org.

Writing your own door call wrappers (Solaris only)

Unlike syscalls or ioctls, door calls transfer data between two userspace programs, albeit through a kernel interface. Programs may use completely proprietary semantics in the data buffers passed between them. Therefore it may not be possible to capture these semantics within a Valgrind door call or door return wrapper.

Nevertheless, for system or well-known door services it would be beneficial to have a door call and a door return wrapper. Writing such wrapper is pretty much the same as writing ioctl wrappers. Please take a few moments to study the following picture depicting how a door client and a door server interact through the kernel interface in a typical scenario:

```
door client thread          kernel       door server thread
invokes door_call()                      invokes door_return()
-------------------------------------------------------------------
                            <----  PRE(sys_door, DOOR_RETURN)
PRE(sys_door, DOOR_CALL)  --->
                            ---->  POST(sys_door, DOOR_RETURN)
```

```
                                   ----> server_procedure()
                                   <----
                         <----  PRE(sys_door, DOOR_RETURN)
POST(sys_door, DOOR_CALL) <---
```

The first PRE(sys_door, DOOR_RETURN) is invoked with data_ptr=NULL
and data_size=0. That's because it has not received any data from
a door call, yet.

Semantics are described by the following functions
in coregring/m_syswrap/syswrap-solaris.c module:
o For a door call wrapper the following attributes of 'params' argument:
  - data_ptr (and associated data_size) as input buffer (request);
     described in door_call_pre_mem_params_data()
  - rbuf (and associated rsize) as output buffer (response);
     described in door_call_post_mem_params_rbuf()
o For a door return wrapper the following parameters:
  - data_ptr (and associated data_size) as input buffer (request);
     described in door_return_post_mem_data()
  - data_ptr (and associated data_size) as output buffer (response);
     described in door_return_pre_mem_data()

There's a default case which may not be correct and you have to write a
more specific case to get the right behaviour. Unless Valgrind's option
'--sim-hints=lax-doors' is specified, the default case also spits a warning.

As above, please create a bug report and attach the patch as described
on http://www.valgrind.org.

6. README_DEVELOPERS

Building and installing it
~~~~~~~~~~~~~~~~~~~~~~~~~~
To build/install from the GIT repository or from a distribution tarball, refer to the section with the same name in README.

Building Valgrind requires autoconf, GNU make and a suitable C compiler.

Building and not installing it
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
To run Valgrind without having to install it, run coregrind/valgrind with the VALGRIND_LIB environment variable set, where <dir> is the root of the source tree (and must be an absolute path). Eg:

```
  VALGRIND_LIB=~/grind/head4/.in_place ~/grind/head4/coregrind/valgrind
```

This allows you to compile and run with "make" instead of "make install", saving you time.

Or, you can use the 'vg-in-place' script which does that for you.

I recommend compiling with "make --quiet" to further reduce the amount of output spewed out during compilation, letting you actually see any errors, warnings, etc.

Building a distribution tarball
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
To build a distribution tarball from the valgrind sources:

```
  make dist
```

In addition to compiling, linking and packaging everything up, the command will also attempt to build the documentation.

If you only want to test whether the generated tarball is complete and runs regression tests successfully, building documentation is not needed.

```
  make dist BUILD_ALL_DOCS=no
```

If you insist on building documentation some embarrassing instructions can be found in docs/README.

Running the regression tests
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Running the regression tests requires GNU sed, python 3.9 or later, gdb, and a suitable C++ compiler

To build and run all the regression tests, run "make [--quiet] regtest".

To run a subset of the regression tests, execute:

```
  perl tests/vg_regtest <name>
```

where <name> is a directory (all tests within will be run) or a single .vgtest test file, or the name of a program which has a like-named .vgtest file. Eg:

```
  perl tests/vg_regtest memcheck
  perl tests/vg_regtest memcheck/tests/badfree.vgtest
  perl tests/vg_regtest memcheck/tests/badfree
```

The details of each vgtest run are logged to individual "vgtest.log" files. These are listed, and non-passing tests detailed, in the test-suite-overall.log file. (Token *.trs and test-suite.log files are also created, for emulating automake-style testsuites, as expected by tools such as bunsen.)

Running the performance tests
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
To build and run all the performance tests, run "make [--quiet] perf".

To run a subset of the performance suite, execute:

```
  perl perf/vg_perf <name>
```

where <name> is a directory (all tests within will be run) or a single .vgperf test file, or the name of a program which has a like-named .vgperf file. Eg:

```
  perl perf/vg_perf perf/
  perl perf/vg_perf perf/bz2.vgperf
  perl perf/vg_perf perf/bz2
```

To compare multiple versions of Valgrind, use the --vg= option multiple times. For example, if you have two Valgrinds next to each other, one in trunk1/ and one in trunk2/, from within either trunk1/ or trunk2/ do this to compare them on all the performance tests:

```
  perl perf/vg_perf --vg=../trunk1 --vg=../trunk2 perf/
```

Writing regression tests
~~~~~~~~~~~~~~~~~~~~~~~~

Each tool has a tests directory containing regression tests. There is also the gdbserver_tests directory at the top level. Test directories may have architecture, OS and and architecture-OS sub-directories for tests that are specific to one architecture, OS or both.

Once you have written a C or C++ executable that performs the required
tests you will have to modify and create several files. The new files that
will need adding are
(a) a .vgtest file, which controls the test
(b) a .stderr.exp file, which is the golden reference for any Valgrind output
    Note that even if Valgrind doesn't produce any output you will need to
    create an empty .stderr.exp file.
(c) [optional] a .stdout.exp file, the golden reference for any output from the
    test executable
(d) [optional] filter files (see the test directories for examples).

Quite often the output will depend on the platform the test is run on.
Getting the tests to be 'portable' can require filtering or adding multiple
.stderr.exp reference files.

If the test only runs under certain conditions like the availability of functions
in libc or C++ standard versions you will need to modify configure.ac in the
top level directory. See AC_CHECK_FUNCS and the various blocks starting with
AC_MSG_CHECKING.

In the test directory, modify Makefile.am. Add to EXTRA_DIST the .vgtest,
.stderr.exp and .stderr.out files. Add any filters to dist_noinst_SCRIPTS.
Add the test executable name to check_PROGRAMS. Try to respect the
formatting and alphabetical ordering of the Makefile.am. For simple C files that is
sufficient. If you needed to add a feature test to configure.ac then you should
use the same condition to add the executable name to check_PROGRAMS. If the
executable uses C++ you need to add exename_SOURCES. If the executable needs
special compilation or link options, use exename_CFLAGS, exename_CXXFLAGS,
exename_LDFLAGS or exename_LDADD. Finally in Makefile.am it's nice not to
have any warnings, even if they were done on purpose. See configure.ac
and various Makefile.am files for examples of using FLAG_W_*.

The vgtest file contains the instructions for running the test. Typically
it will contain (with examples in quotes)
(a) the name of the test executable: "prog: exename"
(b) arguments for the test executable: "args: hello world"
(c) arguments for the Valgrind tool: "vgopts: -q"
(d) [optional] a check for prerequisites: "prereq: ! ../../tests/os_test darwin"
(e) [optional] a filter: "stderr_filter: filter_fdleak"

See tests/vg_regtest for a full description of all possible vgtest directives.

The easiest way to generate the expected files is to run the test. Create empty
files with touch (otherwise the test won't run) then run the test from the
top directory using perl and vg_regtest script (as in the "Running the
regression tests" section. Then copy "tool/tests/newtest.stderr.out" to
"tool/tests/newtest.stderr.exp". It is better to generate the .stdout.exp
file directly from the testcase. You can do that by redirecting stdout to
the expected file, for instance (in the test directory)
    ./newtest arg1 arg2 > newtest.stdout.exp
This is not always possible - sometimes there are tests that depend on the use
of client requests or have imperfect emulation of opcodes.
Make sure that the test runs and passes.

The last file to change is .gitignore in the top directory. Add a new entry,
for example "/tool/tests/newtest".

Check for mistakes in Makefile.am. In the top directory run
```
make post-regtest-checks
```

You should only see
```
...checking makefile consistency
...checking header files and include directives
```
and no messages related to EXTRA_DIST.

Commit access and try branches
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
To get commit access to the valgrind git repository on sourceware
you will have to ask an existing developer and fill in the following
form: https://sourceware.org/cgi-bin/pdw/ps_form.cgi

Every developer with commit access can use try branches. If you want to try a
branch before pushing you can push to a special named try branch as follows:

```
git push origin $BRANCH:users/$USERNAME/try-$BRANCH
```

Where $BRANCH is the branch name and $USERNAME is your user name.

You can see the status of the builders here:
https://builder.sourceware.org/buildbot/#/builders?tags=valgrind-try

The buildbot will also sent the patch author multiple success/failure emails.

Afterwards you can delete the branch again:

```
git push origin :users/$USERNAME/try-$BRANCH
```

Debugging Valgrind with GDB
~~~~~~~~~~~~~~~~~~~~~~~~~~~
To debug the valgrind launcher program (<prefix>/bin/valgrind) just
run it under gdb in the normal way.

Debugging the main body of the valgrind code (and/or the code for
a particular tool) requires a bit more trickery but can be achieved
without too much problem by following these steps:

(1) Set VALGRIND_LAUNCHER to point to the valgrind executable. Eg:

```
export VALGRIND_LAUNCHER=/usr/local/bin/valgrind
```

or for an uninstalled version in a source directory $DIR:

```
export VALGRIND_LAUNCHER=$DIR/coregrind/valgrind
export VALGRIND_LIB=$DIR/.in_place
```

VALGRIND_LIB is where the default.supp and vgpreload_ libraries
are found (which is under /usr/libexec/valgrind for an installed

version).

(2) Run gdb on the tool executable. Eg:

gdb /usr/local/lib/valgrind/lackey-ppc32-linux

or

gdb $DIR/.in_place/memcheck-x86-linux

(3) Do "handle SIGSEGV SIGILL nostop noprint" in GDB to prevent GDB from stopping on a SIGSEGV or SIGILL:

(gdb) handle SIGILL SIGSEGV nostop noprint

If you are using lldb, then the equivalent command is

(lldb) pro hand -p true -s false -n false SIGILL SIGSEGV

(4) Set any breakpoints you want and proceed as normal for gdb. The macro VG_(FUNC) is expanded to vgPlain_FUNC, so If you want to set a breakpoint VG_(do_exec_inner), you could do like this in GDB:

(gdb) b vgPlain_do_exec_inner

Note: This is just an example, for various reasons internal
function names might be renamed or optimized out
(for example when building with --enable-lto).

(5) Run the tool with required options (the --tool option is required for correct setup), e.g.

(gdb) run --tool=lackey pwd

Steps (1)--(3) can be put in a .gdbinit file, but any directory names must be fully expanded (ie. not an environment variable).

A different and possibly easier way is as follows:

(1) Run Valgrind as normal, but add the flag --wait-for-gdb=yes. This puts the tool executable into a wait loop soon after it gains control. This delays startup for a few seconds.

(2) In a different shell, do "gdb /proc/<pid>/exe <pid>", where <pid> you read from the output printed by (1). This attaches GDB to the tool executable, which should be in the above mentioned wait loop.

(3) Do "cont" to continue. After the loop finishes spinning, startup will continue as normal. Note that comment (3) above re passing signals applies here too.

The default build of Valgrind uses "-g -O2". This is OK most of the time, but with sophisticated optimization it can be difficult to

see the contents of variables. A quick way to get to see function
variables is to temporarily add "__attribute__((optnone))" before
the function definition and rebuild. Alternatively modify
Makefile.all.am and remove -O2 from AM_CFLAGS_BASE. That will
require you to reconfigure and rebuild Valgrind.

Self-hosting
~~~~~~~~~~~~
This section explains:
  (A) How to configure Valgrind to run under Valgrind.
      Such a setup is called self hosting, or outer/inner setup.
  (B) How to run Valgrind regression tests in a 'self-hosting' mode,
      e.g. to verify Valgrind has no bugs such as memory leaks.
  (C) How to run Valgrind performance tests in a 'self-hosting' mode,
      to analyse and optimise the performance of Valgrind and its tools.

(A) How to configure Valgrind to run under Valgrind:

(1) Check out 2 trees, "Inner" and "Outer". Inner runs the app
    directly. Outer runs Inner.

(2) Configure Inner with --enable-inner and build as usual.

(3) Configure Outer normally and build+install as usual.
    Note: You must use a "make install"-ed valgrind.
    Do *not* use vg-in-place for the Outer valgrind.

(4) Choose a very simple program (date) and try

```
outer/.../bin/valgrind --sim-hints=enable-outer --trace-children=yes  \
  --smc-check=all-non-file \
  --run-libc-freeres=no --tool=cachegrind -v \
  inner/.../vg-in-place --vgdb-prefix=./inner --tool=none -v prog
```

If you omit the --trace-children=yes, you'll only monitor Inner's launcher
program, not its stage2. Outer needs --run-libc-freeres=no, as otherwise
it will try to find and run __libc_freeres in the inner, while libc is not
used by the inner. Inner needs --vgdb-prefix=./inner to avoid inner
gdbserver colliding with outer gdbserver.
Currently, inner does *not* use the client request
VALGRIND_DISCARD_TRANSLATIONS for the JITted code or the code patched for
translation chaining. So the outer needs --smc-check=all-non-file to
detect the modified code.

Debugging the whole thing might imply to use up to 3 GDB:
  * a GDB attached to the Outer valgrind, allowing
    to examine the state of Outer.
  * a GDB using Outer gdbserver, allowing to
    examine the state of Inner.
  * a GDB using Inner gdbserver, allowing to
    examine the state of prog.

The whole thing is fragile, confusing and slow, but it does work well enough
for you to get some useful performance data. Inner has most of
~~~~~~~~~~~~

its output (ie. those lines beginning with "==<pid>==") prefixed with a '>', which helps a lot. However, when running regression tests in an Outer/Inner setup, this prefix causes the reg test diff to fail. Give --sim-hints=no-inner-prefix to the Inner to disable the production of the prefix in the stdout/stderr output of Inner.

The allocators in coregrind/m_mallocfree.c and VEX/priv/main_util.h are annotated with client requests so Memcheck can be used to find leaks and use after free in an Inner Valgrind.

The Valgrind "big lock" is annotated with helgrind client requests so Helgrind and DRD can be used to find race conditions in an Inner Valgrind.

All this has not been tested much, so don't be surprised if you hit problems.

When using self-hosting with an outer Callgrind tool, use '--pop-on-jump' (on the outer). Otherwise, Callgrind has much higher memory requirements.

(B) Regression tests in an outer/inner setup:

To run all the regression tests with an outer memcheck, do :

```
perl tests/vg_regtest --outer-valgrind=../outer/.../bin/valgrind \
                      --all
```

To run a specific regression tests with an outer memcheck, do:

```
perl tests/vg_regtest --outer-valgrind=../outer/.../bin/valgrind \
                      none/tests/args.vgtest
```

To run regression tests with another outer tool:

```
perl tests/vg_regtest --outer-valgrind=../outer/.../bin/valgrind \
                      --outer-tool=helgrind --all
```

--outer-args allows to give specific arguments to the outer tool, replacing the default one provided by vg_regtest.

Note: --outer-valgrind must be a "make install"-ed valgrind.
Do *not* use vg-in-place.

When an outer valgrind runs an inner valgrind, a regression test produces one additional file <testname>.outer.log which contains the errors detected by the outer valgrind. E.g. for an outer memcheck, it contains the leaks found in the inner, for an outer helgrind or drd, it contains the detected race conditions.

The file tests/outer_inner.supp contains suppressions for the irrelevant or benign errors found in the inner.

A regression test running in the inner (e.g. memcheck/tests/badrw) will cause the inner to report an error, which is expected and checked as usual when running the regtests in an outer/inner setup. However, the outer will often also observe an error, e.g. a jump using uninitialised data, or a read/write outside the bounds of a heap block. When the outer reports such an error, it will output the

inner host stacktrace. To this stacktrace, it will append the
stacktrace of the inner guest program. For example, this is an error
reported by the outer when the inner runs the badrw regtest:

```
  ==8119== Invalid read of size 2
  ==8119==    at 0x7F2EFD7AF: ???
  ==8119==    by 0x7F2C82EAF: ???
  ==8119==    by 0x7F180867F: ???
  ==8119==    by 0x40051D: main (badrw.c:5)
  ==8119==    by 0x7F180867F: ???
  ==8119==    by 0x1BFF: ???
  ==8119==    by 0x3803B7F0: _______VVVVVVVV_appended_inner_guest_stack_VVVVVVVV_______ (m_execontext.c:332)
  ==8119==    by 0x40055C: main (badrw.c:22)
  ==8119==  Address 0x55cd03c is 4 bytes before a block of size 16 alloc'd
  ==8119==    at 0x2804E26D: vgPlain_arena_malloc (m_mallocfree.c:1914)
  ==8119==    by 0x2800BAB4: vgMemCheck_new_block (mc_malloc_wrappers.c:368)
  ==8119==    by 0x2800BC87: vgMemCheck_malloc (mc_malloc_wrappers.c:403)
  ==8119==    by 0x28097EAE: do_client_request (scheduler.c:1861)
  ==8119==    by 0x28097EAE: vgPlain_scheduler (scheduler.c:1425)
  ==8119==    by 0x280A7237: thread_wrapper (syswrap-linux.c:103)
  ==8119==    by 0x280A7237: run_a_thread_NORETURN (syswrap-linux.c:156)
  ==8119==    by 0x3803B7F0: _______VVVVVVVV_appended_inner_guest_stack_VVVVVVVV_______ (m_execontext.c:332)
  ==8119==    by 0x4C294C4: malloc (vg_replace_malloc.c:298)
  ==8119==    by 0x40051D: main (badrw.c:5)
```

In the above, the first stacktrace starts with the inner host stacktrace,
which in this case is some JITted code. Such code sometimes contains IPs
that points in the inner guest code (0x40051D: main (badrw.c:5)).
After the separator, we have the inner guest stacktrace.
The second stacktrace gives the stacktrace where the heap block that was
overrun was allocated. We see it was allocated by the inner valgrind
in the client arena (first part of the stacktrace). The second part is
the guest stacktrace that did the allocation.

(C) Performance tests in an outer/inner setup:

To run all the performance tests with an outer cachegrind, do :

```
   perl perf/vg_perf --outer-valgrind=../outer/.../bin/valgrind perf
```

To run a specific perf test (e.g. bz2) in this setup, do :

```
   perl perf/vg_perf --outer-valgrind=../outer/.../bin/valgrind perf/bz2
```

To run all the performance tests with an outer callgrind, do :

```
   perl perf/vg_perf --outer-valgrind=../outer/.../bin/valgrind \
                     --outer-tool=callgrind perf
```

Note: --outer-valgrind must be a "make install"-ed valgrind.
Do *not* use vg-in-place.

To compare the performance of multiple Valgrind versions, do :

```
   perl perf/vg_perf --outer-valgrind=../outer/.../bin/valgrind \
     --outer-tool=callgrind \
     --vg=../inner_xxxx --vg=../inner_yyyy perf
```

(where inner_xxxx and inner_yyyy are the toplevel directories of
the versions to compare).

```
  Cachegrind and cg_diff are particularly handy to obtain a delta
  between the two versions.

When the outer tool is callgrind or cachegrind, the following
output files will be created for each test:
   <outertoolname>.out.<inner_valgrind_dir>.<tt>.<perftestname>.<pid>
   <outertoolname>.outer.log.<inner_valgrind_dir>.<tt>.<perftestname>.<pid>
 (where tt is the two letters abbreviation for the inner tool(s) run).

For example, the command
    perl perf/vg_perf \
      --outer-valgrind=../outer_trunk/install/bin/valgrind \
      --outer-tool=callgrind \
      --vg=../inner_tchain --vg=../inner_trunk perf/many-loss-records

produces the files
    callgrind.out.inner_tchain.no.many-loss-records.18465
    callgrind.outer.log.inner_tchain.no.many-loss-records.18465
    callgrind.out.inner_tchain.me.many-loss-records.21899
    callgrind.outer.log.inner_tchain.me.many-loss-records.21899
    callgrind.out.inner_trunk.no.many-loss-records.21224
    callgrind.outer.log.inner_trunk.no.many-loss-records.21224
    callgrind.out.inner_trunk.me.many-loss-records.22916
    callgrind.outer.log.inner_trunk.me.many-loss-records.22916


Printing out problematic blocks
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If you want to print out a disassembly of a particular block that
causes a crash, do the following.

Try running with "--vex-guest-chase=no --trace-flags=10000000
--trace-notbelow=999999".  This should print one line for each block
translated, and that includes the address.

Then re-run with 999999 changed to the highest bb number shown.
This will print the one line per block, and also will print a
disassembly of the block in which the fault occurred.


Formatting the code with clang-format
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
clang-format is a tool to format C/C++/... code.  The root directory of the
Valgrind tree contains file .clang-format which is a configuration for this tool
and specifies a style for Valgrind.  This gives you an option to use
clang-format to easily format Valgrind code which you are modifying.

The Valgrind codebase is not globally formatted with clang-format.  It means
that you should not use the tool to format a complete file after making changes
in it because that would lead to creating unrelated modifications.

The right approach is to format only updated or new code.  By using an
integration with a text editor, it is possible to reformat arbitrary blocks
of code with a single keystroke.  Refer to the upstream documentation which
```

describes integration with various editors and IDEs:
https://clang.llvm.org/docs/ClangFormat.html.

Updating zstd
~~~~~~~~~~~~~
Similar to libiberty, we have to import a copy of zstd rather than linking
with a library. There isn't (yet) a script to automate this, so it has to be
done manually.

The version currently in use can be seen in coregrind/m_debuginfo/zstd.h.
Look for ZSTD_VERSION_MAJOR ZSTD_VERSION_MINOR and ZSTD_VERSION_RELEASE.

- Get the source of zstd from

  https://github.com/facebook/zstd

- Checkout the latest release tag (should be vMAJ.MIN.REL)

- Copy {zstd git repo}/lib/zstd.h to coregrind/m_debuginfo/zstd.h

- cd to {zstd git repo}/build/single_file_libs and run ./create_single_file_decoder.sh

- You cannot simply copy and use the generated zstddeclib.c!
  All calls to libc functions in this file need replacing with VG_ versions.
  Merge the newly generated zstddeclib.c with coregrind/m_debuginfo/zstddeclib.c.
  Make sure to keep the copy of the BSD license in the C file.
~~~~~~~~~~~~~

7. README_PACKAGERS

Greetings, packaging person! This information is aimed at people building binary distributions of Valgrind.

Thanks for taking the time and effort to make a binary distribution of Valgrind. The following notes may save you some trouble.

-- If your toolchain (compiler, linker) support lto, using the configure option --enable-lto=yes will produce a smaller/faster valgrind (up to 10%).

-- Do not ship your Linux distro with a completely stripped /lib/ld.so. At least leave the debugging symbol names on -- line number info isn't necessary. If you don't want to leave symbols on ld.so, alternatively you can have your distro install ld.so's debuginfo package by default, or make ld.so.debuginfo be a requirement of your Valgrind RPM/DEB/whatever.

Reason for this is that Valgrind's Memcheck tool needs to intercept calls to, and provide replacements for, some symbols in ld.so at startup (most importantly strlen). If it cannot do that, Memcheck shows a large number of false positives due to the highly optimised strlen (etc) routines in ld.so. This has caused some trouble in the past. As of version 3.3.0, on some targets (ppc32-linux, ppc64-linux), Memcheck will simply stop at startup (and print an error message) if such symbols are not present, because it is infeasible to continue.

It's not like this is going to cost you much space. We only need the symbols for ld.so (a few K at most). Not the debug info and not any debuginfo or extra symbols for any other libraries.

-- (Unfortunate but true) When you configure to build with the --prefix=/foo/bar/xyzzy option, the prefix /foo/bar/xyzzy gets baked into valgrind. The consequence is that you _must_ install valgrind at the location specified in the prefix. If you don't, it may appear to work, but will break doing some obscure things, particularly doing fork() and exec().

So you can't build a relocatable RPM / whatever from Valgrind.

-- Don't strip the debug info off lib/valgrind/$platform/vgpreload*.so in the installation tree. Either Valgrind won't work at all, or it will still work if you do, but will generate less helpful error messages. Here's an example:

Mismatched free() / delete / delete []

```
   at 0x40043249: free (vg_clientfuncs.c:171)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
   Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: __builtin_vec_new (vg_clientfuncs.c:152)
   by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
   by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
   by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

This tells you that some memory allocated with new[] was freed with free().

```
Mismatched free() / delete / delete []
   at 0x40043249: (inside vgpreload_memcheck.so)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
   Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: (inside vgpreload_memcheck.so)
   by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
   by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
   by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

This isn't so helpful. Although you can tell there is a mismatch, the names of the allocating and deallocating functions are no longer visible. The same kind of thing occurs in various other messages from valgrind.

-- Don't strip symbols from libexec/valgrind/* in the installation tree. Doing so will likely cause problems. Removing the line number info is probably OK (at least for some of the files in that directory), although that has not been tested by the Valgrind developers.

One consequence of stripping these binaries is that if Valgrind crashes it won't be able to print out a useful callstack. Here is an example posted on Stack Overflow

```
valgrind: the 'impossible' happened: Killed by fatal signal

host stacktrace:
==7732== at 0x38091C12: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x38050E84: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x380510A9: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x380D4F7B: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x380E3946: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
```

Bug reports like this are less likely to be resolved.

-- Please test the final installation works by running it on something huge. I suggest checking that it can start and exit successfully both Firefox and OpenOffice.org. I use these as test programs, and I know they fairly thoroughly exercise Valgrind. The command lines to use are:

valgrind -v --trace-children=yes firefox

valgrind -v --trace-children=yes soffice

If you find any more hints/tips for packaging, please report
it as a bugreport. See http://www.valgrind.org for details.

8. README.S390

Requirements

- You need GCC 3.4 or later to compile the s390 port.
- To run valgrind a z10 machine or any later model is recommended.
  Older machine models down to and including z990 may work but have
  not been tested extensively.

Limitations

- 31-bit client programs are not supported.
- Hexadecimal floating point is not supported.
- Transactional memory is not supported. The transactional-execution
  facility is masked off from HWCAP.
- A full list of unimplemented instructions can be retrieved from
  `docs/internals/s390-opcodes.csv', by grepping for "not implemented".
- FP signalling is not accurate. E.g., the "compare and signal"
  instructions behave like their non-signalling counterparts.
- On machine models predating z10, cachegrind will assume a z10 cache
  architecture. Otherwise, cachegrind will query the hosts cache system
  and use those parameters.
- Some gcc versions use mvc to copy 4/8 byte values. This will affect
  certain debug messages. For example, memcheck will complain about
  4 one-byte reads/writes instead of just a single read/write.

Hardware facilities

Valgrind does not require that the host machine has the same hardware
facilities as the machine for which the client program was compiled.
This is convenient. If possible, the JIT compiler will translate the
client instructions according to the facilities available on the host.
This means, though, that probing for hardware facilities by issuing
instructions from that facility and observing whether SIGILL is thrown
may not work. As a consequence, programs that attempt to do so may
behave differently. It is believed that this is a rare use case.

Reading Material
----
(1) ELF ABI s390x Supplement
    https://github.com/IBM/s390x-abi/releases
(2) z/Architecture Principles of Operation
    https://publibfp.dhe.ibm.com/epubs/pdf/a227832d.pdf
(3) Collection of z/Architecture publications
    https://linux.mainframe.blog/zarchitecture-principles-of-operation/

9. README.android

How to cross-compile and run on Android. Please read to the end, since there are important details further down regarding crash avoidance and GPU support.

These notes were last updated on 4 Nov 2014, for Valgrind SVN revision 14689/2987.

These instructions are known to work, or have worked at some time in the past, for:

```
arm:
  Android 4.0.3 running on a (rooted, AOSP build) Nexus S.
  Android 4.0.3 running on Motorola Xoom.
  Android 4.0.3 running on android arm emulator.
  Android 4.1   running on android emulator.
  Android 2.3.4 on Nexus S worked at some time in the past.

x86:
  Android 4.0.3 running on android x86 emulator.

mips32:
  Android 4.1.2 running on android mips emulator.
  Android 4.2.2 running on android mips emulator.
  Android 4.3   running on android mips emulator.
  Android 4.0.4 running on BROADCOM bcm7425

arm64:
  Android 4.5 (?) running on ARM Juno
```

On android-arm, GDBserver might insert breaks at wrong addresses. Feedback on this welcome.

Other configurations and toolchains might work, but haven't been tested. Feedback is welcome.

Toolchain:

```
  For arm32, x86 and mips32 you need the android-ndk-r6 native
    development kit.  r6b and r7 give a non-completely-working build;
    see http://code.google.com/p/android/issues/detail?id=23203
    For the android emulator, the versions needed and how to install
    them are described in README.android_emulator.

    You can get android-ndk-r6 from
    http://dl.google.com/android/ndk/android-ndk-r6-linux-x86.tar.bz2

  For arm64 (aarch64) you need the android-ndk-r10c NDK, from
    http://dl.google.com/android/ndk/android-ndk-r10c-linux-x86_64.bin
```

Install the NDK somewhere. Doesn't matter where. Then:

```
# Modify this (obviously). Note, this "export" command is only done
# so as to reduce the amount of typing required. None of the commands
# below read it as part of their operation.
#
export NDKROOT=/path/to/android-ndk-r<version>


# Then cd to the root of your Valgrind source tree.
#
cd /path/to/valgrind/source/tree


# After this point, you don't need to modify anything. Just copy and
# paste the commands below.


# Set up toolchain paths.
#
# For ARM
export AR=$NDKROOT/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-ar
export LD=$NDKROOT/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-ld
export CC=$NDKROOT/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-gcc

# For x86
export AR=$NDKROOT/toolchains/x86-4.4.3/prebuilt/linux-x86/bin/i686-android-linux-ar
export LD=$NDKROOT/toolchains/x86-4.4.3/prebuilt/linux-x86/bin/i686-android-linux-ld
export CC=$NDKROOT/toolchains/x86-4.4.3/prebuilt/linux-x86/bin/i686-android-linux-gcc

# For MIPS32
export AR=$NDKROOT/toolchains/mipsel-linux-android-4.8/prebuilt/linux-x86_64/bin/mipsel-linux-android-ar
export LD=$NDKROOT/toolchains/mipsel-linux-android-4.8/prebuilt/linux-x86_64/bin/mipsel-linux-android-ld
export CC=$NDKROOT/toolchains/mipsel-linux-android-4.8/prebuilt/linux-x86_64/bin/mipsel-linux-android-gcc

# For ARM64 (AArch64)
export AR=$NDKROOT/toolchains/aarch64-linux-android-4.9/prebuilt/linux-x86_64/bin/aarch64-linux-android-ar
export LD=$NDKROOT/toolchains/aarch64-linux-android-4.9/prebuilt/linux-x86_64/bin/aarch64-linux-android-ld
export CC=$NDKROOT/toolchains/aarch64-linux-android-4.9/prebuilt/linux-x86_64/bin/aarch64-linux-android-gcc


# Do configuration stuff. Don't mess with the --prefix in the
# configure command below, even if you think it's wrong.
# You may need to set the --with-tmpdir path to something
# different if /sdcard doesn't work on the device -- this is
# a known cause of difficulties.

# The below re-generates configure, Makefiles, ...
# This is not needed if you start from a release tarball.
./autogen.sh

# for ARM
```

```
CPPFLAGS="--sysroot=$NDKROOT/platforms/android-3/arch-arm" \
  CFLAGS="--sysroot=$NDKROOT/platforms/android-3/arch-arm" \
  ./configure --prefix=/data/local/Inst \
  --host=armv7-unknown-linux --target=armv7-unknown-linux \
  --with-tmpdir=/sdcard
# note: on android emulator, android-14 platform was also tested and works.
# It is not clear what this platform nr really is.

# for x86
CPPFLAGS="--sysroot=$NDKROOT/platforms/android-9/arch-x86" \
  CFLAGS="--sysroot=$NDKROOT/platforms/android-9/arch-x86 -fno-pic" \
  ./configure --prefix=/data/local/Inst \
  --host=i686-android-linux --target=i686-android-linux \
  --with-tmpdir=/sdcard

# for MIPS32
CPPFLAGS="--sysroot=$NDKROOT/platforms/android-18/arch-mips" \
  CFLAGS="--sysroot=$NDKROOT/platforms/android-18/arch-mips" \
  ./configure --prefix=/data/local/Inst \
  --host=mipsel-linux-android --target=mipsel-linux-android \
  --with-tmpdir=/sdcard

# for ARM64 (AArch64)
CPPFLAGS="--sysroot=$NDKROOT/platforms/android-21/arch-arm64" \
  CFLAGS="--sysroot=$NDKROOT/platforms/android-21/arch-arm64" \
  ./configure --prefix=/data/local/Inst \
  --host=aarch64-unknown-linux --target=aarch64-unknown-linux \
  --with-tmpdir=/sdcard


# At the end of the configure run, a few lines of details
# are printed.  Make sure that you see these two lines:
#
# For ARM:
#          Platform variant: android
#     Primary -DVGPV string: -DVGPV_arm_linux_android=1
#
# For x86:
#          Platform variant: android
#     Primary -DVGPV string: -DVGPV_x86_linux_android=1
#
# For mips32:
#          Platform variant: android
#     Primary -DVGPV string: -DVGPV_mips32_linux_android=1
#
# For ARM64 (AArch64):
#          Platform variant: android
#     Primary -DVGPV string: -DVGPV_arm64_linux_android=1
#
# If you see anything else at this point, something is wrong, and
# either the build will fail, or will succeed but you'll get something
# which won't work.
```

```
# Build, and park the install tree in `pwd`/Inst
#
make -j4
make -j4 install DESTDIR=`pwd`/Inst


# To get the install tree onto the device:
# (I don't know why it's not "adb push Inst /data/local", but this
# formulation does appear to put the result in /data/local/Inst.)
#
adb push Inst /


# To run (on the device).  There are two things you need to consider:
#
# (1) if you are running on the Android emulator, Valgrind may crash
#     at startup.  This is because the emulator (for ARM) may not be
#     simulating a hardware TLS register.  To get around this, run
#     Valgrind with:
#       --kernel-variant=android-no-hw-tls
#
# (2) if you are running a real device, you need to tell Valgrind
#     what GPU it has, so Valgrind knows how to handle custom GPU
#     ioctls.  You can choose one of the following:
#       --kernel-variant=android-gpu-sgx5xx     # PowerVR SGX 5XX series
#       --kernel-variant=android-gpu-adreno3xx  # Qualcomm Adreno 3XX series
#     If you don't choose one, the program will still run, but Memcheck
#     may report false errors after the program performs GPU-specific ioctls.
#
# Anyway: to run on the device:
#
/data/local/Inst/bin/valgrind [kernel variant args] [the usual args etc]


# Once you're up and running, a handy modify-V-rebuild-reinstall
# command line (on the host, of course) is
#
mq -j2 && mq -j2 install DESTDIR=`pwd`/Inst && adb push Inst /
#
# where 'mq' is an alias for 'make --quiet'.


# One common cause of runs failing at startup is the inability of
# Valgrind to find a suitable temporary directory.  On the device,
# there doesn't seem to be any one location which we always have
# permission to write to.  The instructions above use /sdcard.  If
# that doesn't work for you, and you're Valgrinding one specific
# application which is already installed, you could try using its
# temporary directory, in /data/data, for example
# /data/data/org.mozilla.firefox_beta.
#
# Using /system/bin/logcat on the device is helpful for diagnosing
# these kinds of problems.
```

10. README.android_emulator

How to install and run an android emulator.

mkdir android # or any other place you prefer
cd android

download java JDK
http://www.oracle.com/technetwork/java/javase/downloads/index.html
download android SDK
http://developer.android.com/sdk/index.html
download android NDK
http://developer.android.com/sdk/ndk/index.html

versions I used:
jdk-7u4-linux-i586.tar.gz
android-ndk-r8-linux-x86.tar.bz2
android-sdk_r18-linux.tgz

install jdk
tar xzf jdk-7u4-linux-i586.tar.gz

install sdk
tar xzf android-sdk_r18-linux.tgz

install ndk
tar xjf android-ndk-r8-linux-x86.tar.bz2

setup PATH to use the installed software:
export SDKROOT=$HOME/android/android-sdk-linux
export PATH=$PATH:$SDKROOT/tools:$SDKROOT/platform-tools
export NDKROOT=$HOME/android/android-ndk-r8

install android platforms you want by starting:
android
(from $SDKROOT/tools)

select the platforms you need
I selected and installed:
Android 4.0.3 (API 15)
Upgraded then to the newer version available:
Android sdk 20
Android platform tools 12

then define a virtual device:
Tools -> Manage AVDs...
I define an AVD Name with 64 Mb SD Card, (4.0.3, api 15)
rest is default

```
# compile and make install Valgrind, following README.android


# Start your android emulator (it takes some time).
# You can use adb shell to get a shell on the device
# and see it is working. Note that I usually get
# one or two time out from adb shell before it works
adb shell

# Once the emulator is ready, push your Valgrind to the emulator:
adb push Inst /


# IMPORTANT: when running Valgrind, you may need give it the flag
#
#    --kernel-variant=android-no-hw-tls
#
# since otherwise it may crash at startup.
# See README.android for details.


# if you need to debug:
# You have on the android side a gdbserver
# on the device side:
gdbserver :1234 your_exe

# on the host side:
adb forward tcp:1234 tcp:1234
$HOME/android/android-ndk-r8/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-gdb your_exe
target remote :1234
```

11. README.mips

Supported platforms

- MIPS32 and MIPS64 platforms are currently supported.
- Both little-endian and big-endian cores are supported.
- MIPS DSP ASE on MIPS32 platforms is supported.

Building V for MIPS

- Native build is available for all supported platforms. The build system expects that native GCC is configured correctly and optimized for the platform. Yet, this may not be the case with some Debian distributions which configure GCC to compile to "mips1" by default. Depending on a target platform, using CFLAGS="-mips32r2", CFLAGS="-mips32" or CFLAGS="-mips64" or CFLAGS="-mips64 -mabi=64" will do the trick and compile Valgrind correctly.

- Use of cross-toolchain is supported as well.
- Example of configure line and additional configure options:

$ ./configure --host=mipsel-linux-gnu --prefix=<path_to_install_directory>

* --host=mips-linux-gnu is necessary only if Valgrind is built on platform other then MIPS, tools for building MIPS application have to be in PATH.

* --host=mips-linux-gnu is necessary if you compile it with cross toolchain compiler for big endian platform.

* --host=mipsel-linux-musl is necessary if you compile it with cross toolchain compiler for little endian platform.

* --host=nanomipseb-linux-gnu is necessary if you compile it with cross toolchain compiler for nanoMIPS big endian platform.

* --host=nanomips-linux-gnu is necessary if you compile it with cross toolchain compiler for nanoMIPS little endian platform.

* --build=mips-linux is needed if you want to build it for MIPS32 on 64-bit MIPS system.

* If you are compiling Valgrind for mips32 with gcc version older then gcc (GCC) 4.5.1, you must specify CFLAGS="-mips32r2 -mplt", e.g.

./configure --prefix=<path_to_install_directory>
CFLAGS="-mips32r2 -mplt"

Limitations

- Some gdb tests will fail when gdb (GDB) older than 7.5 is used and gdb is
  not compiled with '--with-expat=yes'.
- You can not compile tests for DSP ASE if you are using gcc (GCC) older
  then 4.6.1 due to a bug in the toolchain.
- Older GCC may have issues with some inline assembly blocks. Get a toolchain
  based on newer GCC versions, if possible.
- Systems with a mips64 cpu having only o32 libraries will misconfigure in case
  no appropriate architecture flag is specified during configure time.
  Be sure to set either mips32 or mips32r2 as the target architecture in that
  case.
- Some tests can not be compiled for nanoMIPS due to limitations in
  preliminary GCC for nanoMIPS. You can use '-i' switch for building tests.

12. README.riscv64

Status
~~~~~~

The RISC-V port targets the 64-bit RISC-V architecture and the Linux operating system. The port has been tested to work on real hardware and under QEMU.

The following ISA base and extensions are currently supported:

Name	Description	#Instrs	Notes
RV64I	Base instruction set	52/52	
RV64M	Integer multiplication & division	12/13	(1)
RV64A	Atomic	22/22	(2)
RV64F	Single-precision floating-point	30/30	(3)
RV64D	Double-precision floating-point	32/32	
RV64Zicsr	Control & status register	3/6	(4), (5)
RV64Zifencei	Instruction-fetch fence	0/1	(6)
RV64C	Compressed	37/37	

Notes:
(1) MULHSU is not recognized.
(2) LR and SC use the VEX "fallback" method which suffers from the ABA problem.
(3) Operations do not check if the input operands are correctly NaN-boxed.
(4) CSRRWI, CSRRSI and CSRRCI are not recognized.
(5) Only registers fflags, frm and fcsr are accepted.
(6) FENCE.I is not recognized.

Implementation tidying-up/TODO notes
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

* Implement a proper "non-fallback" method for LR and SC instructions.
* Add a check for correct NaN-boxing of 32-bit floating-point operands.
* Optimize instruction selection, in particular make more use of <instr>i variants.
* Optimize handling of floating-point exceptions. Avoid helpers and calculate exception flags using the same instruction which produced an actual result.
* Review register usage by the codegen.
* Avoid re-use of Intel-constants CFIC_IA_SPREL and CFIC_IA_BPREL. Generalize them for all architectures or introduce same CFIC_RISCV64_ variants.
* Get rid of the typedef of vki_modify_ldt_t in include/vki/vki-riscv64-linux.h.
* Review if setup_client_stack() should expose AT_SYSINFO_EHDR to clients.
* Make sure that the final exit sequence in run_a_thread_NORETURN() is not racy in regards to accessing the thread state.

13. README.solaris

Requirements

- You need a recent Solaris-like OS to compile this port. Solaris 11 or
  any illumos-based distribution should work, Solaris 10 is not supported.
  Running `uname -r` has to print '5.11'.
- Recent GCC tools are required, GCC 3 will probably not work. GCC version
  4.5 (or higher) is recommended.
- On Illumos you can install the 'build-essential' metapackage which
  includes GCC and many other developer tools.
- Solaris ld has to be the first linker in the PATH. GNU ld cannot be used.
  There is currently no linker check in the configure script but the linking
  phase fails if GNU ld is used. Recent Solaris/illumos distributions are ok.
- A working combination of autotools is required: aclocal, autoheader,
  automake and autoconf have to be found in the PATH. You should be able to
  install pkg:/developer/build/automake and pkg:/developer/build/autoconf
  packages to fulfil this requirement.
- System header files are required. On Solaris, these can be installed with:
    # pkg install system/header
- GNU make is also required. On Solaris, this can be quickly achieved with:
    $ PATH=/usr/gnu/bin:$PATH; export PATH
- For remote debugging support, working GDB is required (see below).
- For running regression tests, GNU sed, grep, awk, diff are required.
  This can be quickly achieved on Solaris by prepending /usr/gnu/bin to PATH.

Compilation

Please follow the generic instructions in the README file,
in the section 'Building and installing it'.

The configure script detects a canonical host to determine which version of
Valgrind should be built. If the system compiler by default produces 32-bit
binaries then only a 32-bit version of Valgrind will be built. To enable
compilation of both 64-bit and 32-bit versions on such a system, issue the
configure script as follows:
./configure CC='gcc -m64' CXX='g++ -m64'

Oracle Solaris and illumos support

One of the main goal of this port is to support both Oracle Solaris and
illumos kernels. This is a very hard task because Solaris kernel traditionally
does not provide a stable syscall interface and because Valgrind contains
several parts that are closely tied to the underlying kernel. For these
reasons, the port needs to detect which syscall interfaces are present. This
detection cannot be done easily at run time and is currently implemented as
a set of configure tests. This means that a binary version of this port can be
executed only on a kernel that is compatible with a kernel that was used
during the configure and compilation time.

Main currently-known incompatibilities:
- Solaris 11 (released in November 2011) removed a large set of syscalls where
  *at variant of the syscall was also present, for example, open() versus
  openat(AT_FDCWD) [1]
- syscall number for unlinkat() is 76 on Solaris 11, but 65 on illumos [2]
- illumos (in April 2013) changed interface of the accept() and pipe()
  syscalls [3]
- posix_spawn() functionality is backed up by true spawn() syscall on Solaris 11.4
  whereas illumos and Solaris 11.3 leverage vfork()
- illumos and older Solaris use utimesys() syscall whereas newer Solaris
  uses utimensat()

[1] http://docs.oracle.com/cd/E26502_01/html/E28556/gkzlf.html#gkzip
[2] https://www.illumos.org/issues/521
[3] https://github.com/illumos/illumos-gate/commit/5dbfd19ad5fcc2b779f40f80fa05c1bd28fd0b4e

Limitations

- The port is Work-In-Progress, many things may not work or they can be subtly
  broken.
- Coredumps produced by Valgrind do not contain all information available,
  especially microstate accounting and processor bindings.
- Accessing contents of /proc/self/psinfo is not thread-safe. That is because
  Valgrind emulates this file on behalf of the client programs. Entire
  open() - read() - close() sequence on this file needs to be performed
  atomically.
- Fork limitations: vfork() is translated to fork(), forkall() is not
  supported.
- Valgrind does not track definedness of some eflags (OF, SF, ZF, AF, CF, PF)
  individually for each flag. After a syscall is finished, when a carry flag
  is set and defined, all other mentioned flags will be also defined even
  though they might be undefined before making the syscall.
- System call "execve" with a file descriptor which points to a hardlink
  is currently not supported. That is because from the opened file descriptor
  itself it is not possible to reverse map the intended pathname.
  Examples are fexecve(3C) and isaexec(3C).
- Program headers PT_SUNW_SYSSTAT and PT_SUNW_SYSSTAT_ZONE are not supported.
  That is, programs linked with mapfile directive RESERVE_SEGMENT and attribute
  TYPE equal to SYSSTAT or SYSSTAT_ZONE will cause Valgrind exit. It is not
  possible for Valgrind to arrange mapping of a kernel shared page at the
  address specified in the mapfile for the guest application. There is currently
  no such mechanism in Solaris. Hacky workarounds are possible, though.
- When a thread has no stack then all system calls will result in Valgrind
  crash, even though such system calls use just parameters passed in registers.
  This should happen only in pathological situations when a thread is created
  with custom mmap'ed stack and this stack is then unmap'ed during thread
  execution.

Remote debugging support

Solaris port of GDB has a major flaw which prevents remote debugging from

working correctly. Fortunately this flaw has an easy fix [4]. Unfortunately
it is not present in the current GDB 7.6.2. This boils down to several
options:
- Use GDB shipped with Solaris 11.2 which has this flaw fixed.
- Wait until GDB 7.7 becomes available (there won't be other 7.6.x releases).
- Build GDB 7.6.2 with the fix by yourself using the following steps:

```
# pkg install developer/gnu-binutils
$ wget http://ftp.gnu.org/gnu/gdb/gdb-7.6.2.tar.gz
$ gzip -dc gdb-7.6.2.tar.gz | tar xf -
$ cd gdb-7.6.2
$ patch -p1 -i /path/to/valgrind-solaris/solaris/gdb-sol-thread.patch
$ export LIBS="-lncurses"
$ export CC="gcc -m64"
$ ./configure --with-x=no --with-curses --with-libexpat-prefix=/usr/lib
$ gmake && gmake install
```

[4] https://sourceware.org/ml/gdb-patches/2013-12/msg00573.html

TODO list

- Fix few remaining failing tests.
- Add more Solaris-specific tests (especially for the door and spawn
  syscalls).
- Provide better error reporting for various subsyscalls.
- Implement storing of extra register state in signal frame.
- Performance comparison against other platforms.
- Prevent SIGPIPE when writing to a socket (coregrind/m_libcfile.c).
- Implement ticket locking for fair scheduling (--fair-sched=yes).
- Implement support in DRD and Helgrind tools for thr_join() with thread == 0.
- Add support for accessing thread-local variables via gdb (auxprogs/getoff.c).
  Requires research on internal libc TLS representation.
- VEX supports AVX, BMI and AVX2. Investigate if they can be enabled on
  Solaris/illumos.
- Investigate support for more flags in AT_SUN_AUXFLAGS.
- Fix Valgrind crash when a thread has no stack and syswrap-main.c accesses
  all possible syscall parameters. Enable helgrind/tests/stackteardown.c
  to see this in effect. Would require awareness of syscall parameter semantics.
- Correctly print arguments of DW_CFA_ORCL_arg_loc in show_CF_instruction() when
  it is implemented in libdwarf.
- Handle a situation when guest program sets SC_CANCEL_FLG in schedctl and
  Valgrind needs to invoke a syscall on its own.

Summary of Solaris 11 Kernel Interfaces Used

Valgrind uses directly the following kernel interfaces (not exhaustive list).
Then, of course, it has very intimate knowledge of all syscalls, many ioctls
and some door calls because it has wrappers around them.
- Syscalls:
  . clock_gettime
  . close
  . connect
  . execve

. exit
. faccessat
. fcntl
. forksys
. fstatat
. getcwd
. getdents
. geteuid
. getgid
. getgroups
. getpeername
. getpid
. getrlimit
. getsockname
. getsockopt
. gettimeofday
. kill
. lseek
. lwp_create
. lwp_exit
. lwp_self
. lwp_sigqueue
. mknodat
. mmap
. mprotect
. munmap
. openat
. pipe
. pollsys
. pread
. prgpsys
. pwrite
. read
. readlinkat
. renameat
. rt_sigprocmask
. send
. setrlimit
. setsockopt
. sigaction
. sigreturn
. sigtimedwait
. so_socket
. spawn
. uname
. unlinkat
. waitsys
. write
- Signal frames. Valgrind decomposes and synthetizes signal frames.
- Flag sc_sigblock flag in the schedctl structure by replacing
  function block_all_signals() from libc. The replacement emulates lwp_sigmask
  syscall. More details in coregrind/vg_preloaded.c.
- Initial stack layout for the main thread is synthetized.
- procfs agent thread and other procfs commands for manipulating the process.

- mmapobj syscall is emulated because it gets in the way of the address space
  manager's control.

Contacts

Please send bug reports and any questions about the port to:
Ivo Raisr <ivosh@ivosh.net>
Petr Pavlu <setup@dagobah.cz>

14. README.freebsd

Installing from ports or via pkg
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

You can install Valgrind using either

pkg install devel/valgrind

or alternatively from ports (if installed)

cd /usr/ports/devel/valgrind && make install clean

devel/valgrind is updated with official releases of Valgrind, normally
in April and October each year. There is an alternative port,
devel/valgrind-devel which occasionally gets updated from the latest
Valgrind source. If you want to have the latest port, check on
https://www.freshports.org/ to see which is the most recent. If you
want to have the very latest version, you will need to build a copy
from source. See README for instructions on getting the source with git.

Building Valgrind
~~~~~~~~~~~~~~~~~

Install ports for autotools, gmake and python.

```
$ sh autogen.sh
$ ./configure --prefix=/where/ever
$ gmake
$ gmake install
```

If you are using a jail for building, make sure that it is configured so that
"uname -r" returns a string that matches the pattern "XX.Y-*" where XX is the
major version (12, 13, 14 ...) and Y is the minor version (0, 1, 2, 3).

Known Limitations (June 2022)

0. Be aware that if you use a wrapper script and run Valgrind on the wrapper
   script Valgrind may hit restrictions if the wrapper script runs any
   Capsicum enabled applications. Examples of Capsicum enabled applications
   are echo, basename, tee, uniq and wc. It is recommended that you either
   avoid these applications or that you run Valgrind directly on your test
   application.
1. There are some limitations when running Valgrind on code that was compiled
   with clang. These issues are not present with code compiled with GCC.
   a) There may be missing source information concerning variables due
      to DWARF extensions used by GCC.
   b) Code that uses OpenMP will generate spurious errors.
2. vgdb invoker, which uses ptrace, may cause system calls to be
   interrupted. As an example, if the debuggee seems to have be

stuck and you press Ctrl-C in gdb the debuggee may execute
one more statement before stopping and returning control to
gdb.

Notes for Developers
~~~~~~~~~~~~~~~~~~~~

See README_DEVELOPERS, README_MISSING_SYSCALL_OR_IOCTL and docs/*
for more general information for developers.

0. Adding syscalls.

When adding syscalls, you need to look at the manpage and also syscalls.master
(online at
https://github.com/freebsd/freebsd/blob/master/sys/kern/syscalls.master
and for 32bit
https://github.com/freebsd/freebsd/blob/master/sys/compat/freebsd32/syscalls.master

and if you installed the src package there should also be

/usr/src/sys/kern/syscalls.master
and
/usr/src/sys/compat/freebsd32/syscalls.master)

syscalls.master is particularly useful for seeing quickly whether parameters
are inputs or outputs.

The syscall wrappers can vary from trivial to difficult. Fortunately, many are
either trivial (no arguments) or easy (Valgrind just needs to know what memory
is being read or written). Some syscalls, such as those involving process
creation and termination, signals and memory mapping require deeper interaction
with Valgrind.

When you add syscalls you will need to modify several files
  a) include/vki/vki-scnums-freebsd.h
    This file contains one #define for each syscall. The _NR_ prefix (Linux
    style) is used rather than SYS_ for compatibility with the rest of the
    Valgrind source.
  b) coregrind/m_syswrap/priv_syswrap-freebsd.h
    This uses the DECL_TEMPLATE macro to generate declarations for the syscall
    before and after wrappers.
  c) coregrind/m_syswrap/syswrap-freebsd.c
    This is where the bulk of the code resides. Toward the end of the file
    the BSDX_/BSDXY macros are used to generate entries in the table of
    syscalls. BSDX_ is used for wrappers that only have a 'before', BSDXY
    if both wrappers are required. In general, syscalls that have no arguments
    or only input arguments just need a BSDX_ macro (before only). Syscalls
    with output arguments need a BSDXY macro (before and after).
  d) If the syscall uses 64bit arguments (long long) then instead of putting
    the wrapper definitions in syswrap-freebsd.c there will be two kinds of
    definition. A 32bit version with split 64bit arguments for x86 in
    syswrap-x86-freebsd.c and 64bit versions without any splitting for amd64
    and arm64 in syswrap-amd64-freebsd.c/syswrap-arm64-freebsd.c.
    Each long long needs to be split into two ARGs in the x86 version.
~~~~~~~~~~~~~~~~~~~~

The PRE (before) wrapper

Each PRE wrapper always contains the following two macro calls

PRINT. This outputs the syscall name and argument values when Valgrind is
executed with
--trace-syscalls=yes

PRE_READ_REGX. This macro lets Valgrind know about the number and types of the
syscall arguments which allows Valgrind to check that they are initialized.
X is the number of arguments. It is best that the argument names match
the man page, but they must match the types and number of arguments in
syscalls.master. Occasionally there are differences between the two.

If the syscall takes pointers to memory there will be one of the following for
each pointer argument.

PRE_MEM_RASCIIZ for NULL terminated ascii strings.

PRE_MEM_READ for pointers to structures or arrays that are read.

PRE_MEM_WRITE for pointers to structures or arrays that are written.

As a rule, the definitions of structures are copied into vki-freebsd.h
with the vki- prefix. [vki - Valgrind kernel interface; this was done
historically to protect against discrepancies between user include
structure definitions and kernel definitions on Linux].

The POST (after) wrapper

These are much easier.

They just contain a POST_MEM_WRITE macro for each output argument.

1. Frequent causes of problems

- New _umtx_op codes. Valgrind will print "WARNING: _umtx_op unsupported value".
  See syswrap-freebsd.c and add new cases for the new codes.
- Additions to auxv. Depending on the entry it may need to be simply copied
  from the host to the guest, it may need to be modified for the guest or
  it may need to be ignored. See initimg-freebsd.c.
- Versioned ioctls. ioctls such as CAMIOCOMMAND are versioned (with
  CAM_VERSION). When the version number is increased the result is a new
  ioctl ID. That means that the ioctl checking code no longer gets called.
  New versions require updates to the version number and the structs that
  are used. (Backwards compatibility is maintained by adding fixed macros
  like CAM_VERSION_0x19, but these are not currently supported in Valgrind).
- ELF PT_LOAD mappings. Either Valgrind will assert or there will be no source
  information in error reports. See VG_(di_notify_mmap) in debuginfo.c
- Because they contain many deliberate errors the regression tests are prone
  to change with changes of compiler. Liberal use of 'volatile' and

'-Wno-warning-flag' can help - see configure.ac

2. Running regression tests

In order to run all of the regression tests you will need to install
the following packages
gdb
gsed

In addition to running "gmake" you will need to run
"gmake check" to build the regression test exectutables
and "gmake regtest". Again, more details can be seen in
README_DEVELOPERS.

If you want to run the 'nightly' script (see nightly/README.txt)
you will need to install coreutils (for GNU cp) and modify the
nightly/conf/freebsd.* files. The default configuration
sends an e-mail to the valgrind-testresults mailing list.

3. Version specific code

For its own use of syscalls and memory layout Valgrind sometimes needs
to detect which version of FreeBSD it is being built on. Historically
that was done using 'uname' at configure time. It can also be achieved
using the __FreeBSD_version macro (in osreldate.h and sys/param.h).
The former header just defines that macro. To see what changes are associated
with different values of the macro you can search the FreeBSD source and
git history. You can also look at

https://docs.freebsd.org/en/books/porters-handbook/versions/

Feedback
~~~~~~~~

If you find any problems please create a bugzilla report at
https://bugs.kde.org using the Valgrind product.

Alternatively you can use the FreeBSD bugzilla
https://bugs.freebsd.org

Credits
~~~~~~~

Valgrind was originally ported to FreeBSD by Doug Rabson
in 2004.

Paul Floyd (that's me), started looking at this project in late 2018,
took a long pause and then continued in earnest in January 2020.

A big thanks to Nick Briggs for helping with the x86 version.

Kyle Evans and Ed Maste for contributing patches and helping with the
integration with FreeBSD ports.

Prior to 2018 many others have also contributed.

Dimitry Andric
Simon Barner
Roman Bogorodskiy
Rebecca Cran
Bryan Drewery
Brian Fundakowski Feldman
Denis Generalov
Mikolaj Golub
Eugene Kilachkoff
Xin LI
Phil Longstaff
Pav Lucistnik
Conrad Meyer
Julien Nadeau
Frerich Raabe
Doug Rabson
Craig Rodrigues
Tom Russo
Stephen Sanders
Stanislav Sedov
Andrei V. Shetuhin
Niklas Sorensson
Ryan Stone
Jerry Toung
Yuri

GNU Licenses

Table of Contents

1. The GNU General Public License

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS

TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

2. The GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the

Document or a portion of it, either copied verbatim, or with
modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of
the Document that deals exclusively with the relationship of the
publishers or authors of the Document to the Document's overall subject
(or to related matters) and contains nothing that could fall directly
within that overall subject. (Thus, if the Document is in part a
textbook of mathematics, a Secondary Section may not explain any
mathematics.) The relationship could be a matter of historical
connection with the subject or with related matters, or of legal,
commercial, philosophical, ethical or political position regarding
them.

The "Invariant Sections" are certain Secondary Sections whose titles
are designated, as being those of Invariant Sections, in the notice
that says that the Document is released under this License. If a
section does not fit the above definition of Secondary then it is not
allowed to be designated as Invariant. The Document may contain zero
Invariant Sections. If the Document does not identify any Invariant
Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed,
as Front-Cover Texts or Back-Cover Texts, in the notice that says that
the Document is released under this License. A Front-Cover Text may
be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy,
represented in a format whose specification is available to the
general public, that is suitable for revising the document
straightforwardly with generic text editors or (for images composed of
pixels) generic paint programs or (for drawings) some widely available
drawing editor, and that is suitable for input to text formatters or
for automatic translation to a variety of formats suitable for input
to text formatters. A copy made in an otherwise Transparent file
format whose markup, or absence of markup, has been arranged to thwart
or discourage subsequent modification by readers is not Transparent.
An image format is not Transparent if used for any substantial amount
of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain
ASCII without markup, Texinfo input format, LaTeX input format, SGML
or XML using a publicly available DTD, and standard-conforming simple
HTML, PostScript or PDF designed for human modification. Examples of
transparent image formats include PNG, XCF and JPG. Opaque formats
include proprietary formats that can be read and edited only by
proprietary word processors, SGML or XML for which the DTD and/or
processing tools are not generally available, and the
machine-generated HTML, PostScript or PDF produced by some word
processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself,
plus such following pages as are needed to hold, legibly, the material
this License requires to appear in the title page. For works in

formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one

stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this

License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.