Valgrind Documentation Set

Development release 2.1.2 July 18 2004

Table of Contents

Valgrind User's Manual

Valgrind User’s Manual

Table of Contents

Chapter 1. Using and understanding the Valgrind core

This section describes the Valgrind core services, flags and behaviours. That means it is relevant regardless of what particular tool you are using. A point of terminology: most references to "valgrind" in the rest of this section (Section 2) refer to the valgrind core services.

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.

Simply put `valgrind --tool=tool_name` at the start of the command line normally used to run the program. For example, if want to run the command `ls -l` using the heavyweight memory-checking tool Memcheck, issue the command:

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

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 computeroutput locations (if that is appropriate).

Your program is then run on a synthetic x86 CPU provided by the Valgrind core. As new computeroutput is executed for the first time, the core hands the computeroutput to the selected tool. The tool adds its own instrumentation computeroutput to this and hands the result back to the core, which coordinates the continued execution of this instrumented computeroutput.

The amount of instrumentation computeroutput added varies widely between tools. At one end of the scale, Memcheck adds computeroutput to check every memory access and every value computed, increasing the size of the computeroutput at least 12 times, and making it run 25-50 times slower than natively. At the other end of the spectrum, the ultra-trivial "none" tool (a.k.a. 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 computeroutput in your application but also in all supporting dynamically-linked (`.so`-format) libraries, including the GNU C library, the X client libraries, Qt, if you work with KDE, and so on.

If you're using one of the error-detection tools, Valgrind will often detect errors in 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 noo control over that computeroutput. 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 attempts to select suppressions which give reasonable behaviour for the libc and XFree86 versions detected on your machine. To make it easier to write suppressions, you can use the `--gen-suppressions=yes` option which tells Valgrind to print out a suppression for each error that appears, which you can then copy into a suppressions file.

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.

Getting started

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

Another flag 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 whatever it's worth, debugging OpenOffice.org with Memcheck is a bit easier when using this flag.

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.

This paragraph applies only if you plan to use Memcheck: On rare occasions, optimisation levels at `-O2` and above have been observed to generate computeroutput which fools Memcheck into wrongly reporting 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 unmanagably slow, a plausible 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 complaints from Memcheck. All other tools (as far as we know) are unaffected by optimisation level.

Valgrind understands both the older "stabs" debugging format, used by gcc versions prior to 3.1, and the newer DWARF2 format used by gcc 3.1 and later. We continue to refine and debug our debug-info readers, although the majority of effort will naturally enough go into the newer DWARF2 reader.

When you're ready to roll, just run your application as you would normally, but place `valgrind --tool=tool_name` in front of your usual command-line invocation. Note that you should run the real (machine-computeroutput) 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 flag `--trace-children=yes`, but confusion is still likely.

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` flag to Valgrind.

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`.

2. A less intrusive option is to write the commentary to a file, which you specify by `--log-file=filename`. Note carefully that the commentary is **not** written to the file you specify, but instead to one called `filename.pid12345`, if for example the pid of the traced process is 12345. This is helpful when valgrinding a whole tree of processes at once, since it means that each process writes to its own logfile, rather than the result being jumbled up in one big logfile.

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 (I 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 pretty useless 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 two command-line flags:

   - `-e` or `--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.

   - `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.

Reporting of errors

When one of the error-checking tools (Memcheck, Addrcheck, Helgrind) detects something bad happening in the program, an error message is written to the commentary. For example:

```
==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==    by 0x40371E5E: __libc_start_main (libc-start.c:129)
==25832==    by 0x80485D1: (within /home/sewardj/newmat10/bogon)
==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 currently malloc'd or free'd 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 malloc'd/free'd block, for example reading free'd memory, Valgrind reports not only the location where the error happened, but also where the associated block was malloc'd/free'd.

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. If you're using a tool (Memcheck, Addrcheck) which does address checking, and your program attempts to read from address zero, the tool will emit a message to this effect, and the program will then duly 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 here, Valgrind will simply stop collecting errors after 300 different errors have been seen, or 30000 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 300/30000 limits apply after suppressed errors are removed. These limits are defined in `vg_include.h` and can be increased if necessary.

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

Suppressing errors

The error-checking tools detect numerous problems in the base libraries, such as the GNU C library, and the XFree86 client libraries, which come pre-installed on your GNU/Linux system. 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 cooked up 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` flag described in Command-line flags for the Valgrind core [7].

Each error to be suppressed is described very specifically, to minimise the possibility that a suppression-directive inadvertantly 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` flag, at the end of execution, Valgrind prints out one line for each used suppression, giving its name and the number of times it got used. Here's the suppressions used by a run of `valgrind --tool=memcheck ls -l`:

```
--27579-- supp: 1
socketcall.connect(serv_addr)/__libc_connect/__nscd_getgrgid_r
--27579-- supp: 1
socketcall.connect(serv_addr)/__libc_connect/__nscd_getpwuid_r
--27579-- supp: 6 strrchr/_dl_map_object_from_fd/_dl_map_object
```

Multiple suppressions files are allowed. By default, Valgrind uses `$PREFIX/lib/valgrind/default.supp`. You can ask to add suppressions from another file, by specifying `--suppressions=/path/to/file.supp`.

If you want to understand more about suppressions, look at an existing suppressions file whilst reading the following documentation. The file `glibc-2.2.supp`, in the source distribution, provides some good examples.

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 (Nb: no spaces are allowed), eg:

  ```
  tool_name1,tool_name2:suppression_name
  ```

  Recall that Valgrind-2.0.X 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.

  Valgrind's core can detect certain PThreads API errors, for which this line reads:

  ```
  core:PThread
  ```

- 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 four of these lines.

  Locations may be either names of shared objects/executables or wildcards matching function names. They begin `obj:` and `fun:` respectively. Function and object names to match against may use the wildcard characters `*` and `?`.

  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.

- 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 ??? 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
}
```

Suppress 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 with Red Hat 7.2 have had their symbol tables removed.

Note: since the above two examples did not make it clear, you can freely mix the `obj:` and `fun:` styles of description within a single suppression record.

Command-line flags for the Valgrind core

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

You invoke Valgrind like this:

```
valgrind --tool=tool_name [valgrind-options] your-prog [your-prog options]
```

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

Tool-selection option

The single most important option.

- `--tool=name`

  Run the Valgrind tool called *name*, e.g. Memcheck, Addrcheck, Cachegrind, etc.

Basic Options

These options work with all tools.

- `--help`

  Show help for all options, both for the core and for the selected tool.

- `--help-debug`

  Same as `--help`, but also lists debugging options which usually are only of use to 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.

- `-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 flag increases the verbosity level.

- `-q --quiet`

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

- `--trace-children=no` [default]

  `--trace-children=yes`

  When enabled, Valgrind will trace into child processes. This is confusing and usually not what you want, so is disabled by default.

- `--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.

- `--log-file=<filename>`

  Specifies that Valgrind should send all of its messages to the specified file. In fact, the file name used is created by concatenating the text `filename`, ".pid" and the process ID, so as to create a file per process. The specified file name may not be the empty string.

- `--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 [2].

Error-related options

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

- `--demangle=no`

  `--demangle=yes` [default]

  Disable/enable automatic demangling (decoding) of C++ names. Enabled by default. When enabled, Valgrind will attempt to translate encomputeroutputd C++ procedure names back to something approaching the original. The demangler handles symbols mangled by g++ versions 2.X and 3.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 suppressions file contents dependent on the state of Valgrind's demangling machinery, and would also be slow and pointless.

- `--num-callers=<number>` [default=4]

  By default, Valgrind shows four levels of function call names to help you identify program locations. You can change that number with this option. This can help in determining the program's location in deeply-nested call chains. Note that errors are commoned up using only the top three function locations (the place in the current function, and that of its two immediate callers). So this doesn't affect the total number of errors reported.

  The maximum value for this is 50. 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.

- `--error-limit=yes` [default]

  `--error-limit=no`

  When enabled, Valgrind stops reporting errors after 30000 in total, or 300 different ones, have been seen. This is to stop the error tracking machinery from becoming a huge performance overhead in programs with many errors.

- `--show-below-main=yes`

  `--show-below-main=no` [default]

  By default, stack traces for errors do not show any functions that appear beneath `main()`; most of the time it's uninteresting C library stuff. If this option is enabled, these entries below `main()` will be shown.

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

  Specifies an extra file from which to read descriptions of errors to suppress. You may use as many extra suppressions files as you like.

- `--gen-suppressions=no` [default]

  `--gen-suppressions=yes`

  When enabled, Valgrind will pause after every error shown, and print the line: `---- Print suppression ? --- [Return/N/n/Y/y/C/c] ----`

  The prompt's behaviour is the same as for the `--db-attach` option.

  If you choose to, Valgrind will print out 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.

  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, eg. by adding wildcards to function names. Also, 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.

- `--track-fds=no` [default]

  `--track-fds=yes`

  When enabled, Valgrind will print out a list of open file descriptors on exit. Along with each file descriptor, Valgrind prints out 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.

- `--db-attach=no` [default]

  `--db-attach=yes`

  When enabled, Valgrind will pause after every error shown, and print the line: `---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ----`

  Pressing `Ret`, or `N Ret` or `n Ret`, causes Valgrind not to start a debugger for this error.

  `Y Ret` or `y Ret` causes Valgrind to start a debugger, for the program at this point. When you have finished with the debugger, quit from it, and the program will continue. Trying to continue from inside the debugger doesn't work.

  `C Ret` or `c Ret` causes Valgrind not to start a debugger, and not to ask again.

  Note: `--db-attach=yes` conflicts with `--trace-children=yes`. You can't use them together. Valgrind refuses to start up in this situation.

1 May 2002: this is a historical relic which could be easily fixed if it gets in your way. Mail me and complain if this is a problem for you.

Nov 2002: if you're sending output to a logfile or to a network socket, I guess this option doesn't make any sense. Caveat emptor.

- `--db-command=<command>` [default: gdb -nw %f %p]

  This specifies how Valgrind will invoke the debugger. By default it will use whatever GDB is detected at build time, which is usually `/usr/bin/gdb`. Using this command, you can specify some alternative command to invoke the debugger you want to use.

  The command string given can include one or instances of the `%p` and `%f` expansions. Each instance of `%p` expands to the PID of the process to be debugged and each instance of `%f` expands to the path to the executable for the process to be debugged.

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

  When using `--db-attach=yes` and `--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.

`malloc()`-related Options

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

- `--alignment=<number>` [default: 8]

  By default Valgrind's `malloc`, `realloc`, etc, return 4-byte aligned addresses. These are suitable for any accesses on x86 processors. Some programs might however assume that `malloc` et al return 8- or more aligned memory. The supplied value must be between 4 and 4096 inclusive, and must be a power of two.

- `--sloppy-malloc=no` [default]

  `--sloppy-malloc=yes`

  When enabled, all requests for malloc/calloc are rounded up to a whole number of machine words -- in other words, made divisible by 4. For example, a request for 17 bytes of space would result in a 20-byte area being made available. This works around bugs in sloppy libraries which assume that they can safely rely on malloc/calloc requests being rounded up in this fashion. Without the workaround, these libraries tend to generate large numbers of errors when they access the ends of these areas.

  Valgrind snapshots dated 17 Feb 2002 and later are cleverer about this problem, and you should no longer need to use this flag. To put it bluntly, if you do need to use this flag, your program violates the ANSI C semantics defined for `malloc` and `free`, even if it appears to work correctly, and you should fix it, at least if you hope for maximum portability.

Rare Options

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

- `--run-libc-freeres=yes` [default]

  `--run-libc-freeres=no`

  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 and Addrcheck therefore try and run `__libc_freeres` at exit.

  Unfortunately, in some versions of glibc, `__libc_freeres` is sufficiently buggy to cause segmentation faults. This is particularly noticeable on Red Hat 7.1. So this flag 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`.

- `--weird-hacks=hack1,hack2,...`

  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 hacks are enabled. Use with caution! Currently known hacks 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 initialized when writing. Without this, using some device drivers with a large number of strange ioctl commands becomes very tiresome.

- `--signal-polltime=<time>` [default=50]

  How often to poll for signals (in milliseconds). Only applies for older kernels that need signal routing.

- `--lowlat-signals=no` [default]

  `--lowlat-signals=yes`

  Improve wake-up latency when a thread receives a signal.

- `--lowlat-syscalls=no` [default]

  `--lowlat-syscalls=yes`

  Improve wake-up latency when a thread's syscall completes.

Debugging Valgrind Options

There are also some options for debugging Valgrind itself. You shouldn't need to use them in the normal run of things. Nevertheless:

- `--single-step=no` [default]

  `--single-step=yes`

  When enabled, each x86 insn is translated separately into instrumented computeroutput. When disabled, translation is done on a per-basic-block basis, giving much better translations.

- `--optimise=no`

  `--optimise=yes` [default]

  When enabled, various improvements are applied to the intermediate computeroutput, mainly aimed at allowing the simulated CPU's registers to be cached in the real CPU's registers over several simulated instructions.

- `--profile=no`

  `--profile=yes` [default]

  When enabled, does crude internal profiling of Valgrind itself. This is not for profiling your programs. Rather it is to allow the developers to assess where Valgrind is spending its time. The tools must be built for profiling for this to work.

- `--trace-syscalls=no` [default]

  `--trace-syscalls=yes`

  Enable/disable tracing of system call intercepts.

- `--trace-signals=no` [default]

  `--trace-signals=yes`

  Enable/disable tracing of signal handling.

- `--trace-sched=no` [default]

  `--trace-sched=yes`

  Enable/disable tracing of thread scheduling events.

- `--trace-pthread=none` [default]

  `--trace-pthread=some`

  `--trace-pthread=all`

  Specifies amount of trace detail for pthread-related events.

- `--trace-symtab=no` [default]

  `--trace-symtab=yes`

  Enable/disable tracing of symbol table reading.

- `--trace-malloc=no` [default]

  `--trace-malloc=yes`

  Enable/disable tracing of malloc/free (et al) intercepts.

- `--trace-computeroutputgen=XXXXX` [default: 00000]

  Enable/disable tracing of computeroutput generation. Computeroutput can be printed at five different stages of translation; each `X` element must be 0 or 1.

- `--dump-error=<number>` [default: inactive]

  After the program has exited, show gory details of the translation of the basic block containing the <number>'th error context. When used with `--single-step=yes`, can show the exact x86 instruction causing an error. This is all fairly dodgy and doesn't work at all if threads are involved.

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`. The first two are particularly useful for setting the default tool to use.

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`.

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 malloc, free, signals, threads, etc, work, although you don't see that.

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 conversely to make queries. In particular, your program can tell Valgrind about changes in memory range permissions that Valgrind would not otherwise know about, and so allows clients to get Valgrind to do arbitrary custom checks.

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.h`. Tool-specific header files are named after the tool, e.g. `memcheck.h`. All header files can be found in the `include` directory of wherever Valgrind was installed.

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

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).
* 0.60+1em

* 0.60+1em `RUNNING_ON_VALGRIND`:

  returns 1 if running on Valgrind, 0 if running on the real CPU.

* 0.60+1em `VALGRIND_DISCARD_TRANSLATIONS`:

  discard translations of computeroutput in the specified address range. Useful if you are debugging a JITter or some other dynamic computeroutput generation system. After this call, attempts to execute computeroutput in the invalidated address range will cause Valgrind to make new translations of that computeroutput, which is probably the semantics you want. Note that this is implemented naively, and involves checking all 200191 entries in the translation table to see if any of them overlap the specified address range. So try not to call it often, or performance will nosedive. Note that you can be clever about this: you only need to call it when an area which previously contained computeroutput is overwritten with new computeroutput. You can choose to write computeroutput into fresh memory, and just call this occasionally to discard large chunks of old computeroutput all at once.

  Warning: minimally tested, especially for tools other than Memcheck.

* 0.60+1em `VALGRIND_COUNT_ERRORS`:

  returns the number of errors found so far by Valgrind. Can be useful in test harness computeroutput 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.

* 0.60+1em `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.

* 0.60+1em `VALGRIND_FREELIKE_BLOCK`:

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

* 0.60+1em `VALGRIND_CREATE_MEMPOOL`:

  This is similar to `VALGRIND_MALLOCLIKE_BLOCK`, but is tailored towards computeroutput that uses memory pools. See the comments in `valgrind.h` for information on how to use it.

* 0.60+1em `VALGRIND_DESTROY_MEMPOOL`:

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

* 0.60+1em `VALGRIND_MEMPOOL_ALLOC`:

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

* 0.60+1em `VALGRIND_MEMPOOL_FREE`:

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

* 0.60+1em `VALGRIND_NON_SIMD_CALL[0123]`:

  executes a function of 0, 1, 2 or 3 args in the client program on the *real* CPU, not the virtual CPU that Valgrind normally runs computeroutput on. 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.

* 0.60+1em `VALGRIND_PRINTF(format, ...)`:

  printf a message to the log file when running under Valgrind. Nothing is output if not running under Valgrind. Returns the number of characters output.

* 0.60+1em `VALGRIND_PRINTF_BACKTRACE(format, ...)`:

  printf a message to the log file along with a stack backtrace when running under Valgrind. Nothing is output if not running under Valgrind. Returns the number of characters output.

Note that `valgrind.h` is included by all the tool-specific header files (such as `memcheck.h`), so you don't need to include it in your client if you include a tool-specific header.

Support for POSIX Pthreads

Valgrind supports programs which use POSIX pthreads. Getting this to work was technically challenging but it all works well enough for significant threaded applications to work.

It works as follows: threaded apps are (dynamically) linked against `libpthread.so`. Usually this is the one installed with your Linux distribution. Valgrind, however, supplies its own `libpthread.so` and automatically connects your program to it instead.

The fake `libpthread.so` and Valgrind cooperate to implement a user-space pthreads package. This approach avoids the horrible implementation problems of implementing a truly multiprocessor version of Valgrind, but it does mean that threaded apps run only on one CPU, even if you have a multiprocessor machine.

Valgrind schedules your threads in a round-robin fashion, with all threads having equal priority. It switches threads every 50000 basic blocks (typically around 300000 x86 instructions), which means you'll get a much finer interleaving of thread executions than when run natively. This in itself may cause your program to behave differently if you have some kind of concurrency, critical race, locking, or similar, bugs.

As of the Valgrind-1.0 release, the state of pthread support was as follows:

- Mutexes, condition variables, thread-specific data, `pthread_once`, reader-writer locks, semaphores, cleanup stacks, cancellation and thread detaching currently work. Various attribute-like calls are handled but ignored; you get a warning message.
- Currently the following syscalls are thread-safe (nonblocking): `write`, `read`, `nanosleep`, `sleep`, `select`, `poll`, `recvmsg` and `accept`.
- Signals in pthreads are now handled properly(ish): `pthread_sigmask`, `pthread_kill`, `sigwait` and `raise` are now implemented. Each thread has its own signal mask, as POSIX requires. It's a bit kludgey -- there's a system-wide pending signal set, rather than one for each thread. But hey.

Note: As of 18 May 2002, the following threaded programs now work fine on my RedHat 7.2 box: Opera 6.0Beta2, KNode in KDE 3.0, Mozilla-0.9.2.1 and Galeon-0.11.3, both as supplied with RedHat 7.2. Also Mozilla 1.0RC2. OpenOffice 1.0. MySQL 3.something (the current stable release).

Handling of Signals

Valgrind provides suitable handling of signals, so, provided you stick to POSIX stuff, you should be ok. Basic sigaction() and sigprocmask() are handled. Signal handlers may return in the normal way or do longjmp(); both should work ok. As specified by POSIX, a signal is blocked in its own handler. Default actions for signals should work as before. Etc, etc.

Under the hood, dealing with signals is a real pain, and Valgrind's simulation leaves much to be desired. If your program does way-strange stuff with signals, bad things may happen. If so, let us know. We don't promise to fix it, but we'd at least like to be aware of it.

Building and Installing

We use the standard Unix `./configure`, `make`, `make install` mechanism, and we have attempted to ensure that it works on machines with kernel 2.2 or 2.4 and glibc 2.1.X or 2.2.X. I don't think there is much else to say. There are no options apart from the usual `--prefix` that you should give to `./configure`.

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 XFree86 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` ???. It contains some important information.

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

If You Have Problems

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

See Limitations [19] for the known limitations of Valgrind, and for a list of programs which are known not to work on it.

The translator/instrumentor has a lot of assertions in it. They are permanently enabled, and I have no plans to disable them. If one of these breaks, please mail us!

If you get an assertion failure on the expression `chunkSane(ch)` in `vg_free()` in `vg_malloc.c`, this may have happened because your program wrote off the end of a malloc'd block, or before its beginning. Valgrind should have emitted a proper message to that effect before dying in this way. This is a known problem which we should fix.

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

Limitations

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

Valgrind will run x86-GNU/Linux ELF dynamically linked binaries, on a kernel 2.2.X or 2.4.X system, subject to the following constraints:

- No support for 3DNow instructions. If the translator encounters these, Valgrind will generate a SIGILL when the instruction is executed.
- Pthreads support is improving, but there are still significant limitations in that department. See the section above on Pthreads. Note that your program must be dynamically linked against `libpthread.so`, so that Valgrind can substitute its own implementation at program startup time. If you're statically linked against it, things will fail badly.
- Memcheck assumes that the floating point registers are not used as intermediaries in memory-to-memory copies, so it immediately checks definedness of values loaded from memory by floating-point loads. If you want to write computeroutput which copies around possibly-uninitialised values, you must ensure these travel through the integer registers, not the FPU.
- If your program does its own memory management, rather than using malloc/new/free/delete, it should still work, but Valgrind's error checking won't be so effective. If you describe your program's memory management scheme using "client requests" (Section 3.7 of this manual), 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.
- Programs which switch stacks are not well handled. Valgrind does have support for this, but I don't have great faith in it. It's difficult -- there's no cast-iron way to decide whether a large change in %esp is as a result of the program switching stacks, or merely allocating a large object temporarily on the current stack -- yet Valgrind needs to handle the two situations differently.

- x86 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.

- x86 floating point works correctly, but floating-point computeroutput may run even more slowly than integer computeroutput, due to my simplistic approach to FPU emulation.

- Memory consumption of your program is majorly increased whilst running under Valgrind. 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 computeroutput is 14-16 times larger than the original (!) so you can easily end up with 30+ MB of translations when running (eg) a web browser.

Programs which are known not to work are:

- emacs starts up but immediately concludes it is out of memory and aborts. Emacs has it's own memory-management scheme, but I don't understand why this should interact so badly with Valgrind. Emacs works fine if you build it to use the standard malloc/free routines.

Known platform-specific limitations, as of release 1.0.0:

- On Red Hat 7.3, there have been reports of link errors (at program start time) for threaded programs using `__pthread_clock_gettime` and `__pthread_clock_settime`. This appears to be due to `/lib/librt-2.2.5.so` needing them. Unfortunately I do not understand enough about this problem to fix it properly, and I can't reproduce it on my test RedHat 7.3 system. Please mail me if you have more information / understanding.

How It Works -- A Rough Overview

Some gory details, for those with a passion for gory details. You don't need to read this section if all you want to do is use Valgrind. What follows is an outline of the machinery. A more detailed (and somewhat out of date) description is to be found ???.

Getting started

Valgrind is compiled into a shared object, valgrind.so. The shell script valgrind sets the LD_PRELOAD environment variable to point to valgrind.so. This causes the .so to be loaded as an extra library to any subsequently executed dynamically-linked ELF binary, viz, the program you want to debug.

The dynamic linker allows each .so in the process image to have an initialisation function which is run before main(). It also allows each .so to have a finalisation function run after main() exits.

When valgrind.so's initialisation function is called by the dynamic linker, the synthetic CPU to starts up. The real CPU remains locked in valgrind.so for the entire rest of the program, but the synthetic CPU returns from the initialisation function. Startup of the program now continues as usual -- the dynamic linker calls all the other .so's initialisation routines, and eventually runs main(). This all runs on the synthetic CPU, not the real one, but the client program cannot tell the difference.

Eventually main() exits, so the synthetic CPU calls valgrind.so's finalisation function. Valgrind detects this, and uses it as its cue to exit. It prints summaries of all errors detected, possibly checks for memory leaks, and then exits the finalisation routine, but now on the real CPU. The synthetic CPU has now lost control -- permanently -- so the program exits back to the OS on the real CPU, just as it would have done anyway.

On entry, Valgrind switches stacks, so it runs on its own stack. On exit, it switches back. This means that the client program continues to run on its own stack, so we can switch back and forth between running it on the simulated and real CPUs without difficulty. This was an important design decision, because it makes it easy (well, significantly less difficult) to debug the synthetic CPU.

The translation/instrumentation engine

Valgrind does not directly run any of the original program's computeroutput. Only instrumented translations are run. Valgrind maintains a translation table, which allows it to find the translation quickly for any branch target (computeroutput address). If no translation has yet been made, the translator - a just-in-time translator - is summoned. This makes an instrumented translation, which is added to the collection of translations. Subsequent jumps to that address will use this translation.

Valgrind no longer directly supports detection of self-modifying computeroutput. Such checking is expensive, and in practice (fortunately) almost no applications need it. However, to help people who are debugging dynamic computeroutput generation systems, there is a Client Request (basically a macro you can put in your program) which directs Valgrind to discard translations in a given address range. So Valgrind can still work in this situation provided the client tells it when computeroutput has become out-of-date and needs to be retranslated.

The JITter translates basic blocks -- blocks of straight-line-computeroutput -- as single entities. To minimise the considerable difficulties of dealing with the x86 instruction set, x86 instructions are first translated to a RISC-like intermediate computeroutput, similar to sparc computeroutput, but with an infinite number of virtual integer registers. Initially each insn is translated seperately, and there is no attempt at instrumentation.

The intermediate computeroutput is improved, mostly so as to try and cache the simulated machine's registers in the real machine's registers over several simulated instructions. This is often very effective. Also, we try to remove redundant updates of the simulated machines's condition-computeroutput register.

The intermediate computeroutput is then instrumented, giving more intermediate computeroutput. There are a few extra intermediate-computeroutput operations to support instrumentation; it is all refreshingly simple. After instrumentation there is a cleanup pass to remove redundant value checks.

This gives instrumented intermediate computeroutput which mentions arbitrary numbers of virtual registers. A linear-scan register allocator is used to assign real registers and possibly generate spill computeroutput. All of this is still phrased in terms of the intermediate computeroutput. This machinery is inspired by the work of Reuben Thomas (Mite).

Then, and only then, is the final x86 computeroutput emitted. The intermediate computeroutput is carefully designed so that x86 computeroutput can be generated from it without need for spare registers or other inconveniences.

The translations are managed using a traditional LRU-based caching scheme. The translation cache has a default size of about 14MB.

Tracking the Status of Memory

Each byte in the process' address space has nine bits associated with it: one A bit and eight V bits. The A and V bits for each byte are stored using a sparse array, which flexibly and efficiently covers arbitrary parts of the 32-bit address space without imposing significant space or performance overheads for the parts of the address space never visited. The scheme used, and speedup hacks, are described in detail at the top of the source file `coregrind/vg_memory.c`, so you should read that for the gory details.

System calls

All system calls are intercepted. The memory status map is consulted before and updated after each call. It's all rather tiresome. See `coregrind/vg_syscalls.c` for details.

Signals

All system calls to sigaction() and sigprocmask() are intercepted. If the client program is trying to set a signal handler, Valgrind makes a note of the handler address and which signal it is for. Valgrind then arranges for the same signal to be delivered to its own handler.

When such a signal arrives, Valgrind's own handler catches it, and notes the fact. At a convenient safe point in execution, Valgrind builds a signal delivery frame on the client's stack and runs its handler. If the handler longjmp()s, there is nothing more to be said. If the handler returns, Valgrind notices this, zaps the delivery frame, and carries on where it left off before delivering the signal.

The purpose of this nonsense is that setting signal handlers essentially amounts to giving callback addresses to the Linux kernel. We can't allow this to happen, because if it did, signal handlers would run on the real CPU, not the simulated one. This means the checking machinery would not operate during the handler run, and, worse, memory permissions maps would not be updated, which could cause spurious error reports once the handler had returned.

An even worse thing would happen if the signal handler longjmp'd rather than returned: Valgrind would completely lose control of the client program.

Upshot: we can't allow the client to install signal handlers directly. Instead, Valgrind must catch, on behalf of the client, any signal the client asks to catch, and must delivery it to the client on the simulated CPU, not the real one. This involves considerable gruesome fakery; see `coregrind/vg_signals.c` for details.

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 computeroutput 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== loaded 5950 symbols, 142333 line number locations
==25832==
==25832== Invalid read of size 4
==25832==   at 0x8048724: _ZN10BandMatrix6ReSizeEiii (bogon.cpp:45)
==25832==   by 0x80487AF: main (bogon.cpp:66)
==25832==   by 0x40371E5E: __libc_start_main (libc-start.c:129)
==25832==   by 0x80485D1: (within /home/sewardj/newmat10/bogon)
```

```
==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
==25832==
==25832== exiting, did 1881 basic blocks, 0 misses.
==25832== 223 translations, 3626 bytes in, 56801 bytes out.
```

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

Warning Messages You Might See

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

- `More than 50 errors detected.  Subsequent errors will still be recorded, but in less detail than before.`

  After 50 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 50. The 50 constant can be changed by recompiling Valgrind.

- `More than 300 errors detected.  I'm not reporting any more.  Final error counts may be inaccurate.  Go fix your program!`

  After 300 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 300 number is a compile-time constant.

- `Warning:  client switching stacks?`

  Valgrind spotted such a large change in the stack pointer, `%esp`, 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. You may get many bogus error messages following this, if Valgrind guesses wrong. At the moment "large change" is defined as a change of more that 2000000 in the value of the `%esp` (stack pointer) register.

- `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 I have not yet got round to it). 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.