NAME

valgrind − a suite of tools for debugging and profiling programs

SYNOPSIS

valgrind [*valgrind−options*] [*your−program*] [*your−program−options*]

DESCRIPTION

Valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of debugging and profiling tools. The architecture is modular, so that new tools can be created easily and without disturbing the existing structure.

Some of the options described below work with all Valgrind tools, and some only work with a few or one. The section MEMCHECK OPTIONS and those below it describe tool−specific options.

This manual page covers only basic usage and options. For more comprehensive information, please see the HTML documentation on your system: $INSTALL/share/doc/valgrind/html/index.html, or online: http://www.valgrind.org/docs/manual/index.html.

TOOL SELECTION OPTIONS

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.

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|bad> [default: no]
When enabled, Valgrind will track all file descriptor usage. It will produce errors for bad file descriptor usage like closing a file descriptor twice, using a file descriptor (number) that was never created, or passing an already closed file descriptor to a system call. It will also print out a list of open file descriptors on exit. Or on request, via the gdbserver monitor command *v.info open_fds*.

When an error is generated, or when listing the still open file descriptors at exit, a stack backtrace of where the file was opened is printed. If the file descriptor has already been closed, it will also include a backtrace of where it was previously closed. Any error will include details relating to the file descriptor such as the file name or socket details.

Use **all** to also include reporting on stdin, stdout and stderr (or other inherited file descriptors) at exit. If **bad** option is used, then exclude reporting on leaked file descriptors at exit and only produce errors about bad file decriptors usage.

--modify-fds=<no|yes|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. Use **yes** to restrict the feature from the 0/1/2 file descriptors as they're often used for stdout/stderr redirection.

--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.

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.

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.

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*)

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. If you wish to see the list, use the **−−help−debug** option.

MEMCHECK 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.

CACHEGRIND OPTIONS

−−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**.

CALLGRIND OPTIONS

−−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.

−−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**.

−−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.

−−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").

HELGRIND OPTIONS

−−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.

DRD OPTIONS

−−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

−−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.

MASSIF OPTIONS

−−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**.

BBV OPTIONS

−−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.

LACKEY OPTIONS

−−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.

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**[1] .

SEE ALSO

cg_annotate(1), callgrind_annotate(1), callgrind_control(1), ms_print(1), $INSTALL/share/doc/valgrind/html/index.html or http://www.valgrind.org/docs/manual/index.html, **Debugging your program using Valgrind's gdbserver and GDB**[2] **vgdb**[3], **Valgrind monitor commands**[4], **The Commentary**[5], **Scheduling and Multi−Thread Performance**[6], **Cachegrind: a cache and branch−prediction profiler**[7]. **Execution Trees**[8]

AUTHOR

See the AUTHORS file in the valgrind distribution for a comprehensive list of authors.

This manpage was written by Andres Roldan <aroldan@debian.org> and the Valgrind developers.

NOTES

1. Elfutils Debuginfod
   https://sourceware.org/elfutils/Debuginfod.html
2. Debugging your program using Valgrind's gdbserver and GDB
   http://www.valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver
3. vgdb
   http://www.valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.vgdb
4. Valgrind monitor commands
   http://www.valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.valgrind-monitor-commands
5. The Commentary
   http://www.valgrind.org/docs/manual/manual-core.html#manual-core.comment
6. Scheduling and Multi-Thread Performance
   http://www.valgrind.org/docs/manual/manual-core.html#manual-core.pthreads_perf_sched
7. Cachegrind: a cache and branch-prediction profiler
   http://www.valgrind.org/docs/manual/cg-manual.html
8. Execution Trees
   http://www.valgrind.org/docs/manual/manual-core.html#manual-core.xtree