

































































































































































































- `VALGRIND_CREATE_BLOCK` and `VALGRIND_DISCARD`. `VALGRIND_CREATE_BLOCK` takes an address, a number of bytes and a character string. The specified address range is then associated with that string. When Memcheck reports an invalid access to an address in the range, it will describe it in terms of this block rather than in terms of any other block it knows about. Note that the use of this macro does not actually change the state of memory in any way -- it merely gives a name for the range.

At some point you may want Memcheck to stop reporting errors in terms of the block named by `VALGRIND_CREATE_BLOCK`. To make this possible, `VALGRIND_CREATE_BLOCK` returns a "block handle", which is a C `int` value. You can pass this block handle to `VALGRIND_DISCARD`. After doing so, Valgrind will no longer relate addressing errors in the specified range to the block. Passing invalid handles to `VALGRIND_DISCARD` is harmless.

## 4.8. Memory Pools: describing and working with custom allocators

Some programs use custom memory allocators, often for performance reasons. Left to itself, Memcheck is unable to understand the behaviour of custom allocation schemes as well as it understands the standard allocators, and so may miss errors and leaks in your program. What this section describes is a way to give Memcheck enough of a description of your custom allocator that it can make at least some sense of what is happening.

There are many different sorts of custom allocator, so Memcheck attempts to reason about them using a loose, abstract model. We use the following terminology when describing custom allocation systems:

- Custom allocation involves a set of independent "memory pools".
- Memcheck's notion of a memory pool consists of a single "anchor address" and a set of non-overlapping "chunks" associated with the anchor address.
- Typically a pool's anchor address is the address of a book-keeping "header" structure.
- Typically the pool's chunks are drawn from a contiguous "superblock" acquired through the system `malloc` or `mmap`.

Keep in mind that the last two points above say "typically": the Valgrind mempool client request API is intentionally vague about the exact structure of a mempool. There is no specific mention made of headers or superblocks. Nevertheless, the following picture may help elucidate the intention of the terms in the API:



Note that the header and the superblock may be contiguous or discontinuous, and there may be multiple superblocks associated with a single header; such variations are opaque to Memcheck. The API only requires that your allocation scheme can present sensible values of "pool", "addr" and "size".

Typically, before making client requests related to mempools, a client program will have allocated such a header and superblock for their mempool, and marked the superblock NOACCESS using the VALGRIND\_MAKE\_MEM\_NOACCESS client request.

When dealing with mempools, the goal is to maintain a particular invariant condition: that Memcheck believes the unallocated portions of the pool's superblock (including redzones) are NOACCESS. To maintain this invariant, the client program must ensure that the superblock starts out in that state; Memcheck cannot make it so, since Memcheck never explicitly learns about the superblock of a pool, only the allocated chunks within the pool.

Once the header and superblock for a pool are established and properly marked, there are a number of client requests programs can use to inform Memcheck about changes to the state of a mempool:

- `VALGRIND_CREATE_MEMPOOL(pool, rzB, is_zeroed)`: This request registers the address `pool` as the anchor address for a memory pool. It also provides a size `rzB`, specifying how large the redzones placed around chunks allocated from the pool should be. Finally, it provides an `is_zeroed` argument that specifies whether the pool's chunks are zeroed (more precisely: defined) when allocated.

Upon completion of this request, no chunks are associated with the pool. The request simply tells Memcheck that the pool exists, so that subsequent calls can refer to it as a pool.

- `VALGRIND_CREATE_MEMPOOL_EXT(pool, rzB, is_zeroed, flags)`: Create a memory pool with some flags (that can be OR-ed together) specifying extended behaviour. When flags is zero, the behaviour is identical to `VALGRIND_CREATE_MEMPOOL`.
- The flag `VALGRIND_MEMPOOL_METAPPOOL` specifies that the pieces of memory associated with the pool using `VALGRIND_MEMPOOL_ALLOC` will be used by the application as superblocks to dole out `MALLOC_LIKE` blocks using `VALGRIND_MALLOCLIKE_BLOCK`. In other words, a meta pool is a "2 levels" pool: first level is the blocks described by `VALGRIND_MEMPOOL_ALLOC`. The second level blocks are described using `VALGRIND_MALLOCLIKE_BLOCK`. Note that the association between the pool and the second level blocks is implicit: second level blocks will be located inside first level blocks. It is necessary to use the `VALGRIND_MEMPOOL_METAPPOOL` flag for such 2 levels pools, as otherwise valgrind will detect overlapping memory blocks, and will abort execution (e.g. during leak search).
- `VALGRIND_MEMPOOL_AUTO_FREE`. Such a meta pool can also be marked as an 'auto free' pool using the flag `VALGRIND_MEMPOOL_AUTO_FREE`, which must be OR-ed together with the `VALGRIND_MEMPOOL_METAPPOOL`. For an 'auto free' pool, `VALGRIND_MEMPOOL_FREE` will automatically free the second level blocks that are contained inside the first level block freed with `VALGRIND_MEMPOOL_FREE`. In other words, calling `VALGRIND_MEMPOOL_FREE` will cause implicit calls to `VALGRIND_FREELIKE_BLOCK` for all the second level blocks included in the first level block. Note: it is an error to use the `VALGRIND_MEMPOOL_AUTO_FREE` flag without the `VALGRIND_MEMPOOL_METAPPOOL` flag.
- `VALGRIND_DESTROY_MEMPOOL(pool)`: This request tells Memcheck that a pool is being torn down. Memcheck then removes all records of chunks associated with the pool, as well as its record of the pool's existence. While destroying its records of a mempool, Memcheck resets the redzones of any live chunks in the pool to NOACCESS.
- `VALGRIND_MEMPOOL_ALLOC(pool, addr, size)`: This request informs Memcheck that a `size`-byte chunk has been allocated at `addr`, and associates the chunk with the specified `pool`. If the pool was created with nonzero `rzB` redzones, Memcheck will mark the `rzB` bytes before and after the chunk as NOACCESS. If the pool was created with the `is_zeroed` argument set, Memcheck will mark the chunk as DEFINED, otherwise Memcheck will mark the chunk as UNDEFINED.
- `VALGRIND_MEMPOOL_FREE(pool, addr)`: This request informs Memcheck that the chunk at `addr` should no longer be considered allocated. Memcheck will mark the chunk associated with `addr` as NOACCESS, and delete its record of the chunk's existence.
- `VALGRIND_MEMPOOL_TRIM(pool, addr, size)`: This request trims the chunks associated with `pool`. The request only operates on chunks associated with `pool`. Trimming is formally defined as:
  - All chunks entirely inside the range `addr .. (addr+size-1)` are preserved.

- All chunks entirely outside the range `addr..(addr+size-1)` are discarded, as though `VALGRIND_MEMPOOL_FREE` was called on them.
- All other chunks must intersect with the range `addr..(addr+size-1)`; areas outside the intersection are marked as `NOACCESS`, as though they had been independently freed with `VALGRIND_MEMPOOL_FREE`.

This is a somewhat rare request, but can be useful in implementing the type of mass-free operations common in custom LIFO allocators.

- `VALGRIND_MOVE_MEMPOOL(poolA, poolB)`: This request informs Memcheck that the pool previously anchored at address `poolA` has moved to anchor address `poolB`. This is a rare request, typically only needed if you `realloc` the header of a mempool.

No memory-status bits are altered by this request.

- `VALGRIND_MEMPOOL_CHANGE(pool, addrA, addrB, size)`: This request informs Memcheck that the chunk previously allocated at address `addrA` within `pool` has been moved and/or resized, and should be changed to cover the region `addrB..(addrB+size-1)`. This is a rare request, typically only needed if you `realloc` a superblock or wish to extend a chunk without changing its memory-status bits.

No memory-status bits are altered by this request.

- `VALGRIND_MEMPOOL_EXISTS(pool)`: This request informs the caller whether or not Memcheck is currently tracking a mempool at anchor address `pool`. It evaluates to 1 when there is a mempool associated with that address, 0 otherwise. This is a rare request, only useful in circumstances when client code might have lost track of the set of active mempools.

## 4.9. Debugging MPI Parallel Programs with Valgrind

Memcheck supports debugging of distributed-memory applications which use the MPI message passing standard. This support consists of a library of wrapper functions for the `PMPI_*` interface. When incorporated into the application's address space, either by direct linking or by `LD_PRELOAD`, the wrappers intercept calls to `PMPI_Send`, `PMPI_Recv`, etc. They then use client requests to inform Memcheck of memory state changes caused by the function being wrapped. This reduces the number of false positives that Memcheck otherwise typically reports for MPI applications.

The wrappers also take the opportunity to carefully check size and definedness of buffers passed as arguments to MPI functions, hence detecting errors such as passing undefined data to `PMPI_Send`, or receiving data into a buffer which is too small.

Unlike most of the rest of Valgrind, the wrapper library is subject to a BSD-style license, so you can link it into any code base you like. See the top of `mpi/libmpiwrap.c` for license details.

### 4.9.1. Building and installing the wrappers

The wrapper library will be built automatically if possible. Valgrind's configure script will look for a suitable `mpicc` to build it with. This must be the same `mpicc` you use to build the MPI application you want to debug. By default, Valgrind tries `mpicc`, but you can specify a different one by using the configure-time option `--with-mpicc`. Currently the wrappers are only buildable with `mpiccs` which are based on GNU GCC or Intel's C++ Compiler.

Check that the configure script prints a line like this:

```
checking for usable MPI2-compliant mpicc and mpi.h... yes, mpicc
```

If it says `... no`, your `mpicc` has failed to compile and link a test MPI2 program.

If the configure test succeeds, continue in the usual way with `make` and `make install`. The final install tree should then contain `libmpiwrap-<platform>.so`.

Compile up a test MPI program (eg, MPI hello-world) and try this:

```
LD_PRELOAD=$prefix/lib/valgrind/libmpiwrap-<platform>.so \
mpirun [args] $prefix/bin/valgrind ./hello
```

You should see something similar to the following

```
valgrind MPI wrappers 31901: Active for pid 31901
valgrind MPI wrappers 31901: Try MPIWRAP_DEBUG=help for possible options
```

repeated for every process in the group. If you do not see these, there is an build/installation problem of some kind.

The MPI functions to be wrapped are assumed to be in an ELF shared object with soname matching `libmpi.so*`. This is known to be correct at least for Open MPI and Quadrics MPI, and can easily be changed if required.

## 4.9.2. Getting started

Compile your MPI application as usual, taking care to link it using the same `mpicc` that your Valgrind build was configured with.

Use the following basic scheme to run your application on Valgrind with the wrappers engaged:

```
MPIWRAP_DEBUG=[wrapper-args] \
LD_PRELOAD=$prefix/lib/valgrind/libmpiwrap-<platform>.so \
mpirun [mpirun-args] \
$prefix/bin/valgrind [valgrind-args] \
[application] [app-args]
```

As an alternative to `LD_PRELOADing` `libmpiwrap-<platform>.so`, you can simply link it to your application if desired. This should not disturb native behaviour of your application in any way.

## 4.9.3. Controlling the wrapper library

Environment variable `MPIWRAP_DEBUG` is consulted at startup. The default behaviour is to print a starting banner

```
valgrind MPI wrappers 16386: Active for pid 16386
valgrind MPI wrappers 16386: Try MPIWRAP_DEBUG=help for possible options
```

and then be relatively quiet.

You can give a list of comma-separated options in `MPIWRAP_DEBUG`. These are

- `verbose`: show entries/exits of all wrappers. Also show extra debugging info, such as the status of outstanding `MPI_Requests` resulting from uncompleted `MPI_Irecv`s.
- `quiet`: opposite of `verbose`, only print anything when the wrappers want to report a detected programming error, or in case of catastrophic failure of the wrappers.
- `warn`: by default, functions which lack proper wrappers are not commented on, just silently ignored. This causes a warning to be printed for each unwrapped function used, up to a maximum of three warnings per function.
- `strict`: print an error message and abort the program if a function lacking a wrapper is used.

If you want to use Valgrind's XML output facility (`--xml=yes`), you should pass `quiet` in `MPIWRAP_DEBUG` so as to get rid of any extraneous printing from the wrappers.

## 4.9.4. Functions

All MPI2 functions except `MPI_Wtick`, `MPI_Wtime` and `MPI_Pcontrol` have wrappers. The first two are not wrapped because they return a double, which Valgrind's function-wrap mechanism cannot handle (but it could easily be extended to do so). `MPI_Pcontrol` cannot be wrapped as it has variable arity: `int MPI_Pcontrol(const int level, ...)`

Most functions are wrapped with a default wrapper which does nothing except complain or abort if it is called, depending on settings in `MPIWRAP_DEBUG` listed above. The following functions have "real", do-something-useful wrappers:

```
PMPI_Send PMPI_Bsend PMPI_Ssend PMPI_Rsend

PMPI_Recv PMPI_Get_count

PMPI_Isend PMPI_Ibsend PMPI_Issend PMPI_Irsend

PMPI_Irecv
PMPI_Wait PMPI_Waitall
PMPI_Test PMPI_Testall

PMPI_Iprobe PMPI_Probe

PMPI_Cancel

PMPI_Sendrecv

PMPI_Type_commit PMPI_Type_free

PMPI_Pack PMPI_Unpack

PMPI_Bcast PMPI_Gather PMPI_Scatter PMPI_Alltoall
PMPI_Reduce PMPI_Allreduce PMPI_Op_create

PMPI_Comm_create PMPI_Comm_dup PMPI_Comm_free PMPI_Comm_rank PMPI_Comm_size

PMPI_Error_string
PMPI_Init PMPI_Initialized PMPI_Finalize
```

A few functions such as `PMPI_Address` are listed as `HAS_NO_WRAPPER`. They have no wrapper at all as there is nothing worth checking, and giving a no-op wrapper would reduce performance for no reason.

Note that the wrapper library itself can itself generate large numbers of calls to the MPI implementation, especially when walking complex types. The most common functions called are `PMPI_Extent`, `PMPI_Type_get_envelope`, `PMPI_Type_get_contents`, and `PMPI_Type_free`.

## 4.9.5. Types

MPI-1.1 structured types are supported, and walked exactly. The currently supported combinators are `MPI_COMBINER_NAMED`, `MPI_COMBINER_CONTIGUOUS`, `MPI_COMBINER_VECTOR`, `MPI_COMBINER_HVECTOR`, `MPI_COMBINER_INDEXED`, `MPI_COMBINER_HINDEXED` and `MPI_COMBINER_STRUCT`. This should cover all MPI-1.1 types. The mechanism (function `walk_type`) should extend easily to cover MPI2 combinators.

MPI defines some named structured types (`MPI_FLOAT_INT`, `MPI_DOUBLE_INT`, `MPI_LONG_INT`, `MPI_2INT`, `MPI_SHORT_INT`, `MPI_LONG_DOUBLE_INT`) which are pairs of some basic type and a C `int`. Unfortunately the MPI specification makes it impossible to look inside these types and see where the fields are.

Therefore these wrappers assume the types are laid out as `struct { float val; int loc; }` (for `MPI_FLOAT_INT`), etc, and act accordingly. This appears to be correct at least for Open MPI 1.0.2 and for Quadrics MPI.

If `strict` is an option specified in `MPIWRAP_DEBUG`, the application will abort if an unhandled type is encountered. Otherwise, the application will print a warning message and continue.

Some effort is made to mark/check memory ranges corresponding to arrays of values in a single pass. This is important for performance since asking Valgrind to mark/check any range, no matter how small, carries quite a large constant cost. This optimisation is applied to arrays of primitive types (`double`, `float`, `int`, `long`, `long long`, `short`, `char`, and `long double` on platforms where `sizeof(long double) == 8`). For arrays of all other types, the wrappers handle each element individually and so there can be a very large performance cost.

## 4.9.6. Writing new wrappers

For the most part the wrappers are straightforward. The only significant complexity arises with nonblocking receives.

The issue is that `MPI_Irecv` states the `recv` buffer and returns immediately, giving a handle (`MPI_Request`) for the transaction. Later the user will have to poll for completion with `MPI_Wait` etc, and when the transaction completes successfully, the wrappers have to paint the `recv` buffer. But the `recv` buffer details are not presented to `MPI_Wait` -- only the handle is. The library therefore maintains a shadow table which associates uncompleted `MPI_Requests` with the corresponding buffer address/count/type. When an operation completes, the table is searched for the associated address/count/type info, and memory is marked accordingly.

Access to the table is guarded by a (POSIX pthreads) lock, so as to make the library thread-safe.

The table is allocated with `malloc` and never freed, so it will show up in leak checks.

Writing new wrappers should be fairly easy. The source file is `mpi/libmpiwrap.c`. If possible, find an existing wrapper for a function of similar behaviour to the one you want to wrap, and use it as a starting point. The wrappers are organised in sections in the same order as the MPI 1.1 spec, to aid navigation. When adding a wrapper, remember to comment out the definition of the default wrapper in the long list of defaults at the bottom of the file (do not remove it, just comment it out).

## 4.9.7. What to expect when using the wrappers

The wrappers should reduce Memcheck's false-error rate on MPI applications. Because the wrapping is done at the MPI interface, there will still potentially be a large number of errors reported in the MPI implementation below the interface. The best you can do is try to suppress them.

You may also find that the input-side (buffer length/definedness) checks find errors in your MPI use, for example passing too short a buffer to `MPI_Recv`.

Functions which are not wrapped may increase the false error rate. A possible approach is to run with `MPI_DEBUG` containing `warn`. This will show you functions which lack proper wrappers but which are nevertheless used. You can then write wrappers for them.

A known source of potential false errors are the `PMPI_Reduce` family of functions, when using a custom (user-defined) reduction function. In a reduction operation, each node notionally sends data to a "central point" which uses the specified reduction function to merge the data items into a single item. Hence, in general, data is passed between nodes and fed to the reduction function, but the wrapper library cannot mark the transferred data as initialised before it is handed to the reduction function, because all that happens "inside" the `PMPI_Reduce` call. As a result you may see false positives reported in your reduction function.

# 5. Cachegrind: a high-precision tracing profiler

To use this tool, specify `--tool=cachegrind` on the Valgrind command line.

## 5.1. Overview

Cachegrind is a high-precision tracing profiler. It runs slowly, but collects precise and reproducible profiling data. It can merge and diff data from different runs. To expand on these characteristics:

- *Precise.* Cachegrind measures the exact number of instructions executed by your program, not an approximation. Furthermore, it presents the gathered data at the file, function, and line level. This is different to many other profilers that measure approximate execution time, using sampling, and only at the function level.
- *Reproducible.* In general, execution time is a better metric than instruction counts because it's what users perceive. However, execution time often has high variability. When running the exact same program on the exact same input multiple times, execution time might vary by several percent. Furthermore, small changes in a program can change its memory layout and have even larger effects on runtime. In contrast, instruction counts are highly reproducible; for some programs they are perfectly reproducible. This means the effects of small changes in a program can be measured with high precision.

For these reasons, Cachegrind is an excellent complement to time-based profilers.

Cachegrind can annotate programs written in any language, so long as debug info is present to map machine code back to the original source code. Cachegrind has been used successfully on programs written in C, C++, Rust, and assembly.

Cachegrind can also simulate how your program interacts with a machine's cache hierarchy and branch predictor. This simulation was the original motivation for the tool, hence its name. However, the simulations are basic and unlikely to reflect the behaviour of a modern machine. For this reason they are off by default. If you really want cache and branch information, a profiler like `perf` that accesses hardware counters is a better choice.

## 5.2. Using Cachegrind and `cg_annotate`

First, as for normal Valgrind use, you should compile with debugging info (the `-g` option in most compilers). But by contrast with normal Valgrind use, you probably do want to turn optimisation on, since you should profile your program as it will be normally run.

Second, run Cachegrind itself to gather the profiling data.

Third, run `cg_annotate` to get a detailed presentation of that data. `cg_annotate` can combine the results of multiple Cachegrind output files. It can also perform a diff between two Cachegrind output files.

### 5.2.1. Running Cachegrind

To run Cachegrind on a program `prog`, run:

```
valgrind --tool=cachegrind prog
```

The program will execute (slowly). Upon completion, summary statistics that look like this will be printed:

```
==17942== I refs:      8,195,070
```

The `I refs` number is short for "Instruction cache references", which is equivalent to "instructions executed". If you enable the cache and/or branch simulation, additional counts will be shown.

## 5.2.2. Output File

Cachegrind also writes more detailed profiling data to a file. By default this Cachegrind output file is named `cachegrind.out.<pid>` (where `<pid>` is the program's process ID), but its name can be changed with the `--cachegrind-out-file` option. This file is human-readable, but is intended to be interpreted by the accompanying program `cg_annotate`, described in the next section.

The default `.<pid>` suffix on the output file name serves two purposes. First, it means existing Cachegrind output files aren't immediately overwritten. Second, and more importantly, it allows correct profiling with the `--trace-children=yes` option of programs that spawn child processes.

## 5.2.3. Running `cg_annotate`

Before using `cg_annotate`, it is worth widening your window to be at least 120 characters wide if possible, because the output lines can be quite long.

Then run:

```
cg_annotate <filename>
```

on a Cachegrind output file.

## 5.2.4. The Metadata Section

The first part of the output looks like this:

```
-----
-- Metadata
-----
Invocation:      ../cg_annotate concord.cgout
Command:         ./concord ../cg_main.c
Events recorded: Ir
Events shown:    Ir
Event sort order: Ir
Threshold:       0.1%
Annotation:      on
```

It summarizes how Cachegrind and the profiled program were run.

- **Invocation:** the command line used to produce this output.
- **Command:** the command line used to run the profiled program.
- **Events recorded:** which events were recorded. By default, this is `Ir`. More events will be recorded if cache and/or branch simulation is enabled.
- **Events shown:** the events shown, which is a subset of the events gathered. This can be adjusted with the `--show` option.
- **Event sort order:** the sort order used for the subsequent sections. For example, in this case those sections are sorted from highest `Ir` counts to lowest. If there are multiple events, one will be the primary sort event, and then there can be a secondary sort event, tertiary sort event, etc., though more than one is rarely needed. This

order can be adjusted with the `--sort` option. Note that this does *not* specify the order in which the columns appear. That is specified by the "events shown" line (and can be changed with the `--show` option).

- **Threshold:** `cg_annotate` by default omits files and functions with very low counts to keep the output size reasonable. By default `cg_annotate` only shows files and functions that account for at least 0.1% of the primary sort event. The threshold can be adjusted with the `--threshold` option.
- **Annotation:** whether source file annotation is enabled. Controlled with the `--annotate` option.

If cache simulation is enabled, details of the cache parameters will be shown above the "Invocation" line.

## 5.2.5. Global, File, and Function-level Counts

Next comes the summary for the whole program:

```
-----
-- Summary
-----
Ir_____
8,195,070 (100.0%)  PROGRAM TOTALS
```

The `Ir` column label is suffixed with underscores to show the bounds of the columns underneath.

Then comes file:function counts. Here is the first part of that section:

```
-----
-- File:function summary
-----
Ir_____ file:function
< 3,078,746 (37.6%, 37.6%) /home/njn/grind/wsl/cachegrind/concord.c:
  1,630,232 (19.9%)      get_word
    630,918 (7.7%)      hash
    461,095 (5.6%)      insert
    130,560 (1.6%)      add_existing
     91,014 (1.1%)      init_hash_table
     88,056 (1.1%)      create
     46,676 (0.6%)      new_word_node
< 1,746,038 (21.3%, 58.9%) ./malloc/./malloc/malloc.c:
  1,285,938 (15.7%)      _int_malloc
    458,225 (5.6%)      malloc
< 1,107,550 (13.5%, 72.4%) ./libio/./libio/getc.c:getc
<   551,071 (6.7%, 79.1%) ./string/./sysdeps/x86_64/multiarch/strcmp-avx2.S:__strcmp
<   521,228 (6.4%, 85.5%) ./ctype/./include/ctype.h:
  260,616 (3.2%)      __ctype_tolower_loc
  260,612 (3.2%)      __ctype_b_loc
<   468,163 (5.7%, 91.2%) ????:
  468,151 (5.7%)      ???
<   456,071 (5.6%, 96.8%) /usr/include/ctype.h:get_word
```

Each entry covers one file, and one or more functions within that file. If there is only one significant function within a file, as in the first entry, the file and function are shown on the same line separate by a colon. If there are multiple significant functions within a file, as in the third entry, each function gets its own line.

This example involves a small C program, and shows a combination of code from the program itself (including functions like `get_word` and `hash` in the file `concord.c`) as well as code from system libraries, such as functions like `malloc` and `getc`.

Each entry is preceded with a `<`, which can be useful when navigating through the output in an editor, or grepping through results.

The first percentage in each column indicates the proportion of the total event count is covered by this line. The second percentage, which only shows on the first line of each entry, shows the cumulative percentage of all the entries up to and including this one. The entries shown here account for 96.8% of the instructions executed by the program.

The name `???` is used if the file name and/or function name could not be determined from debugging information. If `???` filenames dominate, the program probably wasn't compiled with `-g`. If `???` function names dominate, the program may have had symbols stripped.

After that comes function:file counts. Here is the first part of that section:

```
-----
-- Function:file summary
-----
Ir_____ function:file
> 2,086,303 (25.5%, 25.5%) get_word:
  1,630,232 (19.9%)      /home/njn/grind/wsl/cachegrind/concord.c
   456,071  (5.6%)      /usr/include/ctype.h
> 1,285,938 (15.7%, 41.1%) _int_malloc:./malloc/./malloc/malloc.c
> 1,107,550 (13.5%, 54.7%) getc:./libio/./libio/getc.c
>   630,918  (7.7%, 62.4%) hash:/home/njn/grind/wsl/cachegrind/concord.c
>   551,071  (6.7%, 69.1%) __strcmp_avx2:./string/./sysdeps/x86_64/multiarch/strcmp-a
>   480,248  (5.9%, 74.9%) malloc:
   458,225  (5.6%)      ./malloc/./malloc/malloc.c
    22,023  (0.3%)      ./malloc/./malloc/arena.c
>   468,151  (5.7%, 80.7%) ????:???
>   461,095  (5.6%, 86.3%) insert:/home/njn/grind/wsl/cachegrind/concord.c
```

This is similar to the previous section, but is grouped by functions first and files second. Also, the entry markers are `>` instead of `<`.

You might wonder why this section is needed, and how it differs from the previous section. The answer is inlining. In this example there are two entries demonstrating a function whose code is effectively spread across more than one file: `get_word` and `malloc`. Here is an example from profiling the Rust compiler, a much larger program that uses inlining more:

```
> 30,469,230 (1.3%, 11.1%) <rustc_middle::ty::context::CtxtInterners>::intern_ty:
    10,269,220 (0.5%)      /home/njn/.cargo/registry/src/github.com-1ecc6299db9ec82
    7,696,827 (0.3%)      /home/njn/dev/rust0/compiler/rustc_middle/src/ty/context
    3,858,099 (0.2%)      /home/njn/dev/rust0/library/core/src/cell.rs
```

In this case the compiled function `intern_ty` includes code from three different source files, due to inlining. These should be examined together. Older versions of `cg_annotate` presented this entry as three separate file: function entries, which would typically be intermixed with all the other entries, making it hard to see that they are all really part of the same function.

## 5.2.6. Per-line Counts

By default, a source file is annotated if it contains at least one function that meets the significance threshold. This can be disabled with the `--annotate` option.

To continue the previous example, here is part of the annotation of the file `concord.c`:

```
-----
-- Annotated source file: /home/njn/grind/wsl/cachegrind/docs/concord.c
-----
```

```
Ir_____
```

```

.          /* Function builds the hash table from the given file. */
.          void init_hash_table(char *file_name, Word_Node *table[])
8 (0.0%)   {
.          FILE *file_ptr;
.          Word_Info *data;
2 (0.0%)   int line = 1, i;
.
.          /* Structure used when reading in words and line numbers. */
3 (0.0%)   data = (Word_Info *) create(sizeof(Word_Info));
.
.          /* Initialise entire table to NULL. */
2,993 (0.0%) for (i = 0; i < TABLE_SIZE; i++)
997 (0.0%)   table[i] = NULL;
.
.          /* Open file, check it. */
4 (0.0%)   file_ptr = fopen(file_name, "r");
2 (0.0%)   if (!(file_ptr)) {
.           fprintf(stderr, "Couldn't open '%s'.\n", file_name);
.           exit(EXIT_FAILURE);
.       }
.
.          /* 'Get' the words and lines one at a time from the file, and insert
.          ** into the table one at a time. */
55,363 (0.7%) while ((line = get_word(data, line, file_ptr)) != EOF)
31,632 (0.4%)   insert(data->word, data->line, table);
.
2 (0.0%)   free(data);
2 (0.0%)   fclose(file_ptr);
6 (0.0%)   }
```

Each executed line is annotated with its event counts. Other lines are annotated with a dot. This may be because they contain no executable code, or they contain executable code but were never executed.

You can easily tell if a function is inlined from this output. If it is not inlined, it will have event counts on the lines containing the opening and closing braces. If it is inlined, it will not have event counts on those lines. In the example above, `init_hash_table` does have counts, so you can tell it is not inlined.

Note again that inlining can lead to surprising results. If a function `f` is always inlined, in the `file:function` and `function:file` sections counts will be attributed to the functions it is inlined into, rather than itself. However, if you look at the line-by-line annotations for `f` you'll see the counts that belong to `f`. So it's worth looking for large counts/percentages in the line-by-line annotations.

Sometimes only a small section of a source file is executed. To minimise uninteresting output, Cachegrind only shows annotated lines and lines within a small distance of annotated lines. Gaps are marked with line numbers, for example:

```
(counts and code for line 704)
-- line 375 -----
-- line 514 -----
(counts and code for line 878)
```

The number of lines of context shown around annotated lines is controlled by the `--context` option.

Any significant source files that could not be found are shown like this:

```
-----
-- Annotated source file: ./malloc/./malloc/malloc.c
-----
```

```
Unannotated because one or more of these original files are unreadable:
- ./malloc/./malloc/malloc.c
```

This is common for library files, because libraries are usually compiled with debugging information but the source files are rarely present on a system.

Cachegrind relies heavily on accurate debug info. Sometimes compilers do not map a particular compiled instruction to line number 0, where the 0 represents "unknown" or "none". This is annoying but does happen in practice. `cg_annotate` prints these in the following way:

```
-----
-- Annotated source file: /home/njn/dev/rust0/compiler/rustc_borrowck/src/lib.rs
-----
Ir_____
1,046,746 (0.0%) <unknown (line 0)>
```

Finally, when annotation is performed, the output ends with a summary of how many counts were annotated and unannotated, and why. For example:

```
-----
-- Annotation summary
-----
Ir_____
3,534,817 (43.1%)   annotated: files known & above threshold & readable, line numbers 1
0                 annotated: files known & above threshold & readable, line numbers 1
0                 unannotated: files known & above threshold & two or more non-identical
4,132,126 (50.4%) unannotated: files known & above threshold & unreadable
59,950 (0.7%)     unannotated: files known & below threshold
468,163 (5.7%)    unannotated: files unknown
```

## 5.2.7. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent.

If the output file name (controlled by `--cachegrind-out-file`) does not contain `%p`, then the outputs from the parent and child will be intermingled in a single output file, which will almost certainly make it unreadable by `cg_annotate`.

## 5.2.8. `cg_annotate` Warnings

There are two situations in which `cg_annotate` prints warnings.

- If a source file is more recent than the Cachegrind output file. This is because the information in the Cachegrind output file is only recorded with line numbers, so if the line numbers change at all in the source (e.g. lines added, deleted, swapped), any annotations will be incorrect.
- If information is recorded about line numbers past the end of a file. This can be caused by the above problem, e.g. shortening the source file while using an old Cachegrind output file. If this happens, the figures for the bogus lines are printed anyway (and clearly marked as bogus) in case they are important.

## 5.2.9. Merging Cachegrind Output Files

`cg_annotate` can merge data from multiple Cachegrind output files in a single run. (There is also a program called `cg_merge` that can merge multiple Cachegrind output files into a single Cachegrind output file, but it is now deprecated because `cg_annotate`'s merging does a better job.)

Use it as follows:

```
cg_annotate file1 file2 file3 ...
```

`cg_annotate` computes the sum of these files (effectively `file1 + file2 + file3`), and then produces output as usual that shows the summed counts.

The most common merging scenario is if you want to aggregate costs over multiple runs of the same program, possibly on different inputs.

## 5.2.10. Differencing Cachegrind output files

`cg_annotate` can diff data from two Cachegrind output files in a single run. (There is also a program called `cg_diff` that can diff two Cachegrind output files into a single Cachegrind output file, but it is now deprecated because `cg_annotate`'s differencing does a better job.)

Use it as follows:

```
cg_annotate --diff file1 file2
```

`cg_annotate` computes the difference between these two files (effectively `file2 - file1`), and then produces output as usual that shows the count differences. Note that many of the counts may be negative; this indicates that the counts for the relevant file/function/line are smaller in the second version than those in the first version.

The simplest common scenario is comparing two Cachegrind output files that came from the same program, but on different inputs. `cg_annotate` will do a good job on this without assistance.

A more complex scenario is if you want to compare Cachegrind output files from two slightly different versions of a program that you have sitting side-by-side, running on the same input. For example, you might have `version1/prog.c` and `version2/prog.c`. A straight comparison of the two would not be useful. Because functions are always paired with filenames, a function `f` would be listed as `version1/prog.c:f` for the first version but `version2/prog.c:f` for the second version.

In this case, use the `--mod-filename` option. Its argument is a search-and-replace expression that will be applied to all the filenames in both Cachegrind output files. It can be used to remove minor differences in filenames.

For example, the option `--mod-filename='s/version[0-9]/versionN/'` will suffice for the above example.

Similarly, sometimes compilers auto-generate certain functions and give them randomized names like `T.1234` where the suffixes vary from build to build. You can use the `--mod-funcname` option to remove small differences like these; it works in the same way as `--mod-filename`.

When `--mod-filename` is used to compare two different versions of the same program, `cg_annotate` will not annotate any file that is different between the two versions, because the per-line counts are not reliable in such a case. For example, imagine if `version2/prog.c` is the same as `version1/prog.c` except with an extra blank line at the top of the file. Every single per-line count will have changed. In comparison, the per-file and per-function counts have not changed, and are still very useful for determining differences between programs. You might think that this means every interesting file will be left unannotated, but again inlining means that files that are identical in the two versions can have different counts on many lines.

## 5.2.11. Cache and Branch Simulation

Cachegrind can simulate how your program interacts with a machine's cache hierarchy and/or branch predictor. The cache simulation models a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). For these machines (in the cases where Cachegrind can auto-detect the cache configuration) Cachegrind simulates the first-level and last-level caches. Therefore, Cachegrind always refers to the I1, D1 and LL (last-level) caches.

When simulating the cache, with `--cache-sim=yes`, Cachegrind gathers the following statistics:

- I cache reads (`Ir`, which equals the number of instructions executed), I1 cache read misses (`I1mr`) and LL cache instruction read misses (`ILmr`).
- D cache reads (`Dr`, which equals the number of memory reads), D1 cache read misses (`D1mr`), and LL cache data read misses (`DLmr`).
- D cache writes (`Dw`, which equals the number of memory writes), D1 cache write misses (`D1mw`), and LL cache data write misses (`DLmw`).

Note that D1 total accesses is given by `D1mr + D1mw`, and that LL total accesses is given by `ILmr + DLmr + DLmw`.

When simulating the branch predictor, with `--branch-sim=yes`, Cachegrind gathers the following statistics:

- Conditional branches executed (`Bc`) and conditional branches mispredicted (`Bcm`).
- Indirect branches executed (`Bi`) and indirect branches mispredicted (`Bim`).

When cache and/or branch simulation is enabled, `cg_annotate` will print multiple counts per line of output. For example:

	<code>Ir</code>		<code>Bc</code>		<code>Bcm</code>		<code>Bi</code>	
>	8,547 (0.1%, 99.4%)		936 (0.1%, 99.1%)		177 (0.3%, 96.7%)		59 (0.0%, 99.9%)	
	8,503 (0.1%)		928 (0.1%)		175 (0.3%)		59 (0.0%)	

## 5.3. Cachegrind Command-line Options

Cachegrind-specific options are:

`--cachegrind-out-file=<file>`

Write the Cachegrind output file to `file` rather than to the default output file, `cachegrind.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

```
--cache-sim=no|yes [no]
```

Enables or disables collection of cache access and miss counts.

```
--branch-sim=no|yes [no]
```

Enables or disables collection of branch instruction and misprediction counts.

```
--instr-at-start=no|yes [yes]
```

Enables or disables instrumentation at the start of execution. Use this in combination with `CACHEGRIND_START_INSTRUMENTATION` and `CACHEGRIND_STOP_INSTRUMENTATION` to measure only part of a client program's execution.

```
--I1=<size>,<associativity>,<line size>
```

Specify the size, associativity and line size of the level 1 instruction cache. Only useful with `--cache-sim=yes`.

```
--D1=<size>,<associativity>,<line size>
```

Specify the size, associativity and line size of the level 1 data cache. Only useful with `--cache-sim=yes`.

```
--LL=<size>,<associativity>,<line size>
```

Specify the size, associativity and line size of the last-level cache. Only useful with `--cache-sim=yes`.

## 5.4. cg\_annotate Command-line Options

```
-h --help
```

Show the help message.

```
--version
```

Show the version number.

```
--diff
```

Diff two Cachegrind output files.

```
--mod-filename <regex> [default: none]
```

Specifies an `s/old/new/` search-and-replace expression that is applied to all filenames. Useful when differencing, for removing minor differences in paths between two different versions of a program that are sitting in different directories. An `i` suffix makes the regex case-insensitive, and a `g` suffix makes it match multiple times.

```
--mod-funcname <regex> [default: none]
```

Like `--mod-filename`, but for filenames. Useful for removing minor differences in randomized names of auto-generated functions generated by some compilers.

```
--show=A,B,C [default: all, using order in the Cachegrind output file]
```

Specifies which events to show (and the column order). Default is to use all present in the Cachegrind output file (and use the order in the file). Best used in conjunction with `--sort`.

```
--sort=A,B,C [default: order in the Cachegrind output file]
```

Specifies the events upon which the sorting of the file:function and function:file entries will be based.

`--threshold=X` [default: 0.1%]

Sets the significance threshold for the file:function and function:files sections. A file or function is shown if it accounts for more than X% of the counts for the primary sort event. If annotating source files, this also affects which files are annotated.

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

When enabled, a percentage is printed next to all event counts. This helps gauge the relative importance of each function and line.

`--annotate`, `--no-annotate`, `--auto=<no|yes>` [default: yes]

Enables or disables source file annotation.

`--context=N` [default: 8]

The number of lines of context to show before and after each annotated line. Use a large number (e.g. 100000) to show all source lines.

## 5.5. cg\_merge Command-line Options

`-o outfile`

Write the output to `outfile` instead of standard output.

## 5.6. cg\_diff Command-line Options

`-h --help`

Show the help message.

`--version`

Show the version number.

`--mod-filename=<expr>` [default: none]

Specifies an `s/old/new/` search-and-replace expression that is applied to all filenames.

`--mod-funcname=<expr>` [default: none]

Like `--mod-filename`, but for filenames.

## 5.7. Cachegrind Client Requests

Cachegrind provides the following client requests in `cachegrind.h`.

`CACHEGRIND_START_INSTRUMENTATION`

Start Cachegrind instrumentation if not already enabled. Use this in combination with `CACHEGRIND_STOP_INSTRUMENTATION` and `--instr-at-start` to measure only part of a client program's execution.

`CACHEGRIND_STOP_INSTRUMENTATION`

Stop Cachegrind instrumentation if not already disabled. Use this in combination with `CACHEGRIND_START_INSTRUMENTATION` and `--instr-at-start` to measure only part of a client program's execution.

## 5.8. Simulation Details

This section talks about details you don't need to know about in order to use Cachegrind, but may be of interest to some people.

### 5.8.1. Cache Simulation Specifics

The cache simulation approximates the hardware of an AMD Athlon CPU circa 2002. Its specific characteristics are as follows:

- Write-allocate: when a write miss occurs, the block written to is brought into the D1 cache. Most modern caches have this property.
- Bit-selection hash function: the set of line(s) in the cache to which a memory block maps is chosen by the middle bits  $M--(M+N-1)$  of the byte address, where:
  - line size =  $2^M$  bytes
  - (cache size / line size / associativity) =  $2^N$  bytes
- Inclusive LL cache: the LL cache typically replicates all the entries of the L1 caches, because fetching into L1 involves fetching into LL first (this does not guarantee strict inclusiveness, as lines evicted from LL still could reside in L1). This is standard on Pentium chips, but AMD Opterons, Athlons and Durons use an exclusive LL cache that only holds blocks evicted from L1. Ditto most modern VIA CPUs.

The cache configuration simulated (cache size, associativity and line size) is determined automatically using the x86 CPUID instruction. If you have a machine that (a) doesn't support the CPUID instruction, or (b) supports it in an early incarnation that doesn't give any cache information, then Cachegrind will fall back to using a default configuration (that of a model 3/4 Athlon). Cachegrind will tell you if this happens. You can manually specify one, two or all three levels (I1/D1/LL) of the cache from the command line using the `--I1`, `--D1` and `--LL` options. For cache parameters to be valid for simulation, the number of sets (with associativity being the number of cache lines in each set) has to be a power of two.

On PowerPC platforms Cachegrind cannot automatically determine the cache configuration, so you will need to specify it with the `--I1`, `--D1` and `--LL` options.

Other noteworthy behaviour:

- References that straddle two cache lines are treated as follows:
  - If both blocks hit --> counted as one hit
  - If one block hits, the other misses --> counted as one miss.
  - If both blocks miss --> counted as one miss (not two)
- Instructions that modify a memory location (e.g. `inc` and `dec`) are counted as doing just a read, i.e. a single data reference. This may seem strange, but since the write can never cause a miss (the read guarantees the block is in the cache) it's not very interesting.

Thus it measures not the number of times the data cache is accessed, but the number of times a data cache miss could occur.

If you are interested in simulating a cache with different properties, it is not particularly hard to write your own cache simulator, or to modify the existing ones in `cg_sim.c`.

### 5.8.2. Branch Simulation Specifics

Cachegrind simulates branch predictors intended to be typical of mainstream desktop/server processors of around 2004.

Conditional branches are predicted using an array of 16384 2-bit saturating counters. The array index used for a branch instruction is computed partly from the low-order bits of the branch instruction's address and partly using the taken/not-taken behaviour of the last few conditional branches. As a result the predictions for any specific branch depend both on its own history and the behaviour of previous branches. This is a standard technique for improving prediction accuracy.

For indirect branches (that is, jumps to unknown destinations) Cachegrind uses a simple branch target address predictor. Targets are predicted using an array of 512 entries indexed by the low order 9 bits of the branch instruction's address. Each branch is predicted to jump to the same address it did last time. Any other behaviour causes a mispredict.

More recent processors have better branch predictors, in particular better indirect branch predictors. Cachegrind's predictor design is deliberately conservative so as to be representative of the large installed base of processors which pre-date widespread deployment of more sophisticated indirect branch predictors. In particular, late model Pentium 4s (Prescott), Pentium M, Core and Core 2 have more sophisticated indirect branch predictors than modelled by Cachegrind.

Cachegrind does not simulate a return stack predictor. It assumes that processors perfectly predict function return addresses, an assumption which is probably close to being true.

See Hennessy and Patterson's classic text "Computer Architecture: A Quantitative Approach", 4th edition (2007), Section 2.3 (pages 80-89) for background on modern branch predictors.

### 5.8.3. Accuracy

Cachegrind's instruction counting has one shortcoming on x86/amd64:

- When a REP-prefixed instruction executes each iteration is counted separately. In contrast, hardware counters count each such instruction just once, no matter how many times it iterates. It is arguable that Cachegrind's behaviour is more useful.

Cachegrind's cache profiling has a number of shortcomings:

- It doesn't account for kernel activity. The effect of system calls on the cache and branch predictor contents is ignored.
- It doesn't account for other process activity. This is arguably desirable when considering a single program.
- It doesn't account for virtual-to-physical address mappings. Hence the simulation is not a true representation of what's happening in the cache. Most caches and branch predictors are physically indexed, but Cachegrind simulates caches using virtual addresses.
- It doesn't account for cache misses not visible at the instruction level, e.g. those arising from TLB misses, or speculative execution.
- Valgrind will schedule threads differently from how they would be when running natively. This could warp the results for threaded programs.
- The x86/amd64 instructions `bts`, `btr` and `btc` will incorrectly be counted as doing a data read if both the arguments are registers, e.g.:

```
btsl %eax, %edx
```

This should only happen rarely.

- x86/amd64 FPU instructions with data sizes of 28 and 108 bytes (e.g. `fsave`) are treated as though they only access 16 bytes. These instructions seem to be rare so hopefully this won't affect accuracy much.

Another thing worth noting is that results are very sensitive. Changing the size of the executable being profiled, or the sizes of any of the shared libraries it uses, or even the length of their file names, can perturb the results. Variations will be small, but don't expect perfectly repeatable results if your program changes at all.

Many Linux distributions perform address space layout randomisation (ASLR), in which identical runs of the same program have their shared libraries loaded at different locations, as a security measure. This also perturbs the results.

## 5.9. Implementation Details

This section talks about details you don't need to know about in order to use Cachegrind, but may be of interest to some people.

### 5.9.1. How Cachegrind Works

The best reference for understanding how Cachegrind works is chapter 3 of "Dynamic Binary Analysis and Instrumentation", by Nicholas Nethercote. It is available on the [Valgrind publications page](#).

### 5.9.2. Cachegrind Output File Format

The file format is fairly straightforward, basically giving the cost centre for every line, grouped by files and functions. It's also totally generic and self-describing, in the sense that it can be used for any events that can be counted on a line-by-line basis, not just cache and branch predictor events. For example, earlier versions of Cachegrind didn't have a branch predictor simulation. When this was added, the file format didn't need to change at all. So the format (and consequently, `cg_annotate`) could be used by other tools.

The file format:

```
file      ::= desc_line* cmd_line events_line data_line+ summary_line
desc_line ::= "desc:" ws? non_nl_string
cmd_line  ::= "cmd:" ws? cmd
events_line ::= "events:" ws? (event ws)+
data_line ::= file_line | fn_line | count_line
file_line  ::= "fl=" filename
fn_line    ::= "fn=" fn_name
count_line ::= line_num (ws+ count)* ws*
summary_line ::= "summary:" ws? count (ws+ count)+ ws*
count      ::= num
```

Where:

- `non_nl_string` is any string not containing a newline.
- `cmd` is a string holding the command line of the profiled program.
- `event` is a string containing no whitespace.
- `filename` and `fn_name` are strings.
- `num` and `line_num` are decimal numbers.
- `ws` is whitespace.

The contents of the "desc:" lines are printed out at the top of the summary. This is a generic way of providing simulation specific information, e.g. for giving the cache configuration for cache simulation.

More than one line of info can be present for each file/fn/line number. In such cases, the counts for the named events will be accumulated.

The number of counts in each line and the `summary_line` should not exceed the number of events in the `event_line`. If the number in each line is less, `cg_annotate` treats those missing as though they were a "0" entry. This can reduce file size.

A `file_line` changes the current file name. A `fn_line` changes the current function name. A `count_line` contains counts that pertain to the current filename/fn\_name. A "fn=" `file_line` and a `fn_line` must appear before any `count_lines` to give the context of the first `count_lines`.

Similarly, each `file_line` must be immediately followed by a `fn_line`.

The summary line is redundant, because it just holds the total counts for each event. But this serves as a useful sanity check of the data; if the totals for each event don't match the summary line, something has gone wrong.

# 6. Callgrind: a call-graph generating cache and branch prediction profiler

To use this tool, you must specify `--tool=callgrind` on the Valgrind command line.

## 6.1. Overview

Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls. Optionally, cache simulation and/or branch prediction (similar to Cachegrind) can produce further information about the runtime behavior of an application.

The profile data is written out to a file at program termination. For presentation of the data, and interactive control of the profiling, two command line tools are provided:

### **callgrind\_annotate**

This command reads in the profile data, and prints a sorted lists of functions, optionally with source annotation.

For graphical visualization of the data, try [KCachegrind](#), which is a KDE/Qt based GUI that makes it easy to navigate the large amount of data that Callgrind produces.

### **callgrind\_control**

This command enables you to interactively observe and control the status of a program currently running under Callgrind's control, without stopping the program. You can get statistics information as well as the current stack trace, and you can request zeroing of counters or dumping of profile data.

## 6.1.1. Functionality

Cachegrind collects flat profile data: event counts (data reads, cache misses, etc.) are attributed directly to the function they occurred in. This cost attribution mechanism is called *self* or *exclusive* attribution.

Callgrind extends this functionality by propagating costs across function call boundaries. If function `foo` calls `bar`, the costs from `bar` are added into `foo`'s costs. When applied to the program as a whole, this builds up a picture of so called *inclusive* costs, that is, where the cost of each function includes the costs of all functions it called, directly or indirectly.

As an example, the inclusive cost of `main` should be almost 100 percent of the total program cost. Because of costs arising before `main` is run, such as initialization of the run time linker and construction of global C++ objects, the inclusive cost of `main` is not exactly 100 percent of the total program cost.

Together with the call graph, this allows you to find the specific call chains starting from `main` in which the majority of the program's costs occur. Caller/callee cost attribution is also useful for profiling functions called from multiple call sites, and where optimization opportunities depend on changing code in the callers, in particular by reducing the call count.

Callgrind's cache simulation is based on that of Cachegrind. Read the documentation for [Cachegrind: a cache and branch-prediction profiler](#) first. The material below describes the features supported in addition to Cachegrind's features.

Callgrind's ability to detect function calls and returns depends on the instruction set of the platform it is run on. It works best on x86 and amd64, and unfortunately currently does not work so well on PowerPC, ARM, Thumb or MIPS code. This is because there are no explicit call or return instructions in these instruction sets, so Callgrind has to rely on heuristics to detect calls and returns.

## 6.1.2. Basic Usage

As with Cachegrind, you probably want to compile with debugging info (the `-g` option) and with optimization turned on.

To start a profile run for a program, execute:

```
valgrind --tool=callgrind [callgrind options] your-program [program options]
```

While the simulation is running, you can observe execution with:

```
callgrind_control -b
```

This will print out the current backtrace. To annotate the backtrace with event counts, run

```
callgrind_control -e -b
```

After program termination, a profile data file named `callgrind.out.<pid>` is generated, where *pid* is the process ID of the program being profiled. The data file contains information about the calls made in the program among the functions executed, together with **Instruction Read (Ir)** event counts.

To generate a function-by-function summary from the profile data file, use

```
callgrind_annotate [options] callgrind.out.<pid>
```

This summary is similar to the output you get from a Cachegrind run with `cg_annotate`: the list of functions is ordered by exclusive cost of functions, which also are the ones that are shown. Important for the additional features of Callgrind are the following two options:

- `--inclusive=yes`: Instead of using exclusive cost of functions as sorting order, use and show inclusive cost.
- `--tree=both`: Interleave into the top level list of functions, information on the callers and the callees of each function. In these lines, which represents executed calls, the cost gives the number of events spent in the call. Indented, above each function, there is the list of callers, and below, the list of callees. The sum of events in calls to a given function (caller lines), as well as the sum of events in calls from the function (callee lines) together with the self cost, gives the total inclusive cost of the function.

By default, you will also get annotated source code for all relevant functions for which the source can be found. In addition to source annotation as produced by `cg_annotate`, you will see the annotated call sites with call counts. For all other options, consult the (Cachegrind) documentation for `cg_annotate`.

For better call graph browsing experience, it is highly recommended to use [KCachegrind](#). If your code has a significant fraction of its cost in *cycles* (sets of functions calling each other in a recursive manner), you have to use KCachegrind, as `callgrind_annotate` currently does not do any cycle detection, which is important to get correct results in this case.

If you are additionally interested in measuring the cache behavior of your program, use Callgrind with the option `--cache-sim=yes`. For branch prediction simulation, use `--branch-sim=yes`. Expect a further slow down approximately by a factor of 2.

If the program section you want to profile is somewhere in the middle of the run, it is beneficial to *fast forward* to this section without any profiling, and then enable profiling. This is achieved by using the command line option `--instr-atstart=no` and running, in a shell: `callgrind_control -i` on just before the interesting code section is executed. To exactly specify the code position where profiling should start, use the client request [CALLGRIND\\_START\\_INSTRUMENTATION](#).

If you want to be able to see assembly code level annotation, specify `--dump-instr=yes`. This will produce profile data at instruction granularity. Note that the resulting profile data can only be viewed with KCachegrind. For assembly annotation, it also is interesting to see more details of the control flow inside of functions, i.e. (conditional) jumps. This will be collected by further specifying `--collect-jumps=yes`.

## 6.2. Advanced Usage

### 6.2.1. Multiple profiling dumps from one program run

Sometimes you are not interested in characteristics of a full program run, but only of a small part of it, for example execution of one algorithm. If there are multiple algorithms, or one algorithm running with different input data, it may even be useful to get different profile information for different parts of a single program run.

Profile data files have names of the form

```
callgrind.out.pid.part-threadID
```

where *pid* is the PID of the running program, *part* is a number incremented on each dump (".part" is skipped for the dump at program termination), and *threadID* is a thread identification ("-threadID" is only used if you request dumps of individual threads with `--separate-threads=yes`).

There are different ways to generate multiple profile dumps while a program is running under Callgrind's supervision. Nevertheless, all methods trigger the same action, which is "dump all profile information since the last dump or program start, and zero cost counters afterwards". To allow for zeroing cost counters without dumping, there is a second action "zero all cost counters now". The different methods are:

- **Dump on program termination.** This method is the standard way and doesn't need any special action on your part.
- **Spontaneous, interactive dumping.** Use

```
callgrind_control -d [hint [PID/Name]]
```

to request the dumping of profile information of the supervised application with PID or Name. *hint* is an arbitrary string you can optionally specify to later be able to distinguish profile dumps. The control program will not terminate before the dump is completely written. Note that the application must be actively running for detection of the dump command. So, for a GUI application, resize the window, or for a server, send a request.

If you are using [KCachegrind](#) for browsing of profile information, you can use the toolbar button **Force dump**. This will request a dump and trigger a reload after the dump is written.

- **Periodic dumping after execution of a specified number of basic blocks.** For this, use the command line option `--dump-every-bb=count`.
- **Dumping at enter/leave of specified functions.** Use the option `--dump-before=function` and `--dump-after=function`. To zero cost counters before entering a function, use `--zero-before=function`.

You can specify these options multiple times for different functions. Function specifications support wildcards: e.g. use `--dump-before='foo*'` to generate dumps before entering any function starting with *foo*.

- **Program controlled dumping.** Insert `CALLGRIND_DUMP_STATS;` at the position in your code where you want a profile dump to happen. Use `CALLGRIND_ZERO_STATS;` to only zero profile counters. See [Client request reference](#) for more information on Callgrind specific client requests.

If you are running a multi-threaded application and specify the command line option `--separate-threads=yes`, every thread will be profiled on its own and will create its own profile dump. Thus, the last two methods will only generate one dump of the currently running thread. With the other methods, you will get multiple dumps (one for each thread) on a dump request.

### 6.2.2. Limiting the range of collected events

By default, whenever events are happening (such as an instruction execution or cache hit/miss), Callgrind is aggregating them into event counters. However, you may be interested only in what is happening within a given

function or starting from a given program phase. To this end, you can disable event aggregation for uninteresting program parts. While attribution of events to functions as well as producing separate output per program phase can be done by other means (see previous section), there are two benefits by disabling aggregation. First, this is very fine-granular (e.g. just for a loop within a function). Second, disabling event aggregation for complete program phases allows to switch off time-consuming cache simulation and allows Callgrind to progress at much higher speed with an slowdown of around factor 2 (identical to `valgrind --tool=none`).

There are two aspects which influence whether Callgrind is aggregating events at some point in time of program execution. First, there is the *collection state*. If this is off, no aggregation will be done. By changing the collection state, you can control event aggregation at a very fine granularity. However, there is not much difference in regard to execution speed of Callgrind. By default, collection is switched on, but can be disabled by different means (see below). Second, there is the *instrumentation mode* in which Callgrind is running. This mode either can be on or off. If instrumentation is off, no observation of actions in the program will be done and thus, no actions will be forwarded to the simulator which could trigger events. In the end, no events will be aggregated. The huge benefit is the much higher speed with instrumentation switched off. However, this only should be used with care and in a coarse fashion: every mode change resets the simulator state (ie. whether a memory block is cached or not) and flushes Valgrinds internal cache of instrumented code blocks, resulting in latency penalty at switching time. Also, cache simulator results directly after switching on instrumentation will be skewed due to identified cache misses which would not happen in reality (if you care about this warm-up effect, you should make sure to temporarily have collection state switched off directly after turning instrumentation mode on). However, switching instrumentation state is very useful to skip larger program phases such as an initialization phase. By default, instrumentation is switched on, but as with the collection state, can be changed by various means.

Callgrind can start with instrumentation mode switched off by specifying option `--instr-atstart=no`. Afterwards, instrumentation can be controlled in two ways: first, interactively with:

```
callgrind_control -i on
```

(and switching off again by specifying "off" instead of "on"). Second, instrumentation state can be programmatically changed with the macros `CALLGRIND_START_INSTRUMENTATION;` and `CALLGRIND_STOP_INSTRUMENTATION;`.

Similarly, the collection state at program start can be switched off by `--instr-atstart=no`. During execution, it can be controlled programmatically with the macro `CALLGRIND_TOGGLE_COLLECT;`. Further, you can limit event collection to a specific function by using `--toggle-collect=function`. This will toggle the collection state on entering and leaving the specified function. When this option is in effect, the default collection state at program start is "off". Only events happening while running inside of the given function will be collected. Recursive calls of the given function do not trigger any action. This option can be given multiple times to specify different functions of interest.

### 6.2.3. Counting global bus events

For access to shared data among threads in a multithreaded code, synchronization is required to avoid raced conditions. Synchronization primitives are usually implemented via atomic instructions. However, excessive use of such instructions can lead to performance issues.

To enable analysis of this problem, Callgrind optionally can count the number of atomic instructions executed. More precisely, for x86/x86\_64, these are instructions using a lock prefix. For architectures supporting LL/SC, these are the number of SC instructions executed. For both, the term "global bus events" is used.

The short name of the event type used for global bus events is "Ge". To count global bus events, use `--collect-bus=yes`.

### 6.2.4. Avoiding cycles

Informally speaking, a cycle is a group of functions which call each other in a recursive way.

Formally speaking, a cycle is a nonempty set  $S$  of functions, such that for every pair of functions  $F$  and  $G$  in  $S$ , it is possible to call from  $F$  to  $G$  (possibly via intermediate functions) and also from  $G$  to  $F$ . Furthermore,  $S$  must

be maximal -- that is, be the largest set of functions satisfying this property. For example, if a third function `H` is called from inside `S` and calls back into `S`, then `H` is also part of the cycle and should be included in `S`.

Recursion is quite usual in programs, and therefore, cycles sometimes appear in the call graph output of Callgrind. However, the title of this chapter should raise two questions: What is bad about cycles which makes you want to avoid them? And: How can cycles be avoided without changing program code?

Cycles are not bad in itself, but tend to make performance analysis of your code harder. This is because inclusive costs for calls inside of a cycle are meaningless. The definition of inclusive cost, i.e. self cost of a function plus inclusive cost of its callees, needs a topological order among functions. For cycles, this does not hold true: callees of a function in a cycle include the function itself. Therefore, KCachegrind does cycle detection and skips visualization of any inclusive cost for calls inside of cycles. Further, all functions in a cycle are collapsed into artificial functions called like `Cycle 1`.

Now, when a program exposes really big cycles (as is true for some GUI code, or in general code using event or callback based programming style), you lose the nice property to let you pinpoint the bottlenecks by following call chains from `main`, guided via inclusive cost. In addition, KCachegrind loses its ability to show interesting parts of the call graph, as it uses inclusive costs to cut off uninteresting areas.

Despite the meaningless of inclusive costs in cycles, the big drawback for visualization motivates the possibility to temporarily switch off cycle detection in KCachegrind, which can lead to misleading visualization. However, often cycles appear because of unlucky superposition of independent call chains in a way that the profile result will see a cycle. Neglecting uninteresting calls with very small measured inclusive cost would break these cycles. In such cases, incorrect handling of cycles by not detecting them still gives meaningful profiling visualization.

It has to be noted that currently, **callgrind\_annotate** does not do any cycle detection at all. For program executions with function recursion, it e.g. can print nonsense inclusive costs way above 100%.

After describing why cycles are bad for profiling, it is worth talking about cycle avoidance. The key insight here is that symbols in the profile data do not have to exactly match the symbols found in the program. Instead, the symbol name could encode additional information from the current execution context such as recursion level of the current function, or even some part of the call chain leading to the function. While encoding of additional information into symbols is quite capable of avoiding cycles, it has to be used carefully to not cause symbol explosion. The latter imposes large memory requirement for Callgrind with possible out-of-memory conditions, and big profile data files.

A further possibility to avoid cycles in Callgrind's profile data output is to simply leave out given functions in the call graph. Of course, this also skips any call information from and to an ignored function, and thus can break a cycle. Candidates for this typically are dispatcher functions in event driven code. The option to ignore calls to a function is `--fn-skip=function`. Aside from possibly breaking cycles, this is used in Callgrind to skip trampoline functions in the PLT sections for calls to functions in shared libraries. You can see the difference if you profile with `--skip-plt=no`. If a call is ignored, its cost events will be propagated to the enclosing function.

If you have a recursive function, you can distinguish the first 10 recursion levels by specifying `--separate-recs10=function`. Or for all functions with `--separate-recs=10`, but this will give you much bigger profile data files. In the profile data, you will see the recursion levels of "func" as the different functions with names "func", "func'2", "func'3" and so on.

If you have call chains "`A > B > C`" and "`A > C > B`" in your program, you usually get a "false" cycle "`B <> C`". Use `--separate-callers2=B --separate-callers2=C`, and functions "B" and "C" will be treated as different functions depending on the direct caller. Using the apostrophe for appending this "context" to the function name, you get "`A > B'A > C'B`" and "`A > C'A > B'C`", and there will be no cycle. Use `--separate-callers=2` to get a 2-caller dependency for all functions. Note that doing this will increase the size of profile data files.

## 6.2.5. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent. To start with empty profile counter values in the child, the client request `CALLGRIND_ZERO_STATS` can be inserted into code to be executed by the child, directly after `fork`.

However, you will have to make sure that the output file format string (controlled by `--callgrind-out-file`) does contain `%p` (which is true by default). Otherwise, the outputs from the parent and child will overwrite each other or will be intermingled, which almost certainly is not what you want.

You will be able to control the new child independently from the parent via `callgrind_control`.

## 6.3. Callgrind Command-line Options

In the following, options are grouped into classes.

Some options allow the specification of a function/symbol name, such as `--dump-before=function`, or `--fn-skip=function`. All these options can be specified multiple times for different functions. In addition, the function specifications actually are patterns by supporting the use of wildcards `*` (zero or more arbitrary characters) and `?` (exactly one arbitrary character), similar to file name globbing in the shell. This feature is important especially for C++, as without wildcard usage, the function would have to be specified in full extent, including parameter signature.

### 6.3.1. Dump creation options

These options influence the name and format of the profile data files.

`--callgrind-out-file=<file>`

Write the profile data to `file` rather than to the default output file, `callgrind.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`. When multiple dumps are made, the file name is modified further; see below.

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

This specifies that event counting should be performed at source line granularity. This allows source annotation for sources which are compiled with debug information (`-g`).

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

This specifies that event counting should be performed at per-instruction granularity. This allows for assembly code annotation. Currently the results can only be displayed by KCachegrind.

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

This option influences the output format of the profile data. It specifies whether strings (file and function names) should be identified by numbers. This shrinks the file, but makes it more difficult for humans to read (which is not recommended in any case).

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

This option influences the output format of the profile data. It specifies whether numerical positions are always specified as absolute values or are allowed to be relative to previous numbers. This shrinks the file size.

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

When enabled, when multiple profile data parts are to be generated these parts are appended to the same output file. Not recommended.

### 6.3.2. Activity options

These options specify when actions relating to event counts are to be executed. For interactive control use `callgrind_control`.

`--dump-every-bb=<count> [default: 0, never]`

Dump profile data every `count` basic blocks. Whether a dump is needed is only checked when Valgrind's internal scheduler is run. Therefore, the minimum setting useful is about 100000. The count is a 64-bit value to make long dump periods possible.

`--dump-before=<function>`

Dump when entering function.

`--zero-before=<function>`

Zero all costs when entering function.

`--dump-after=<function>`

Dump when leaving function.

### 6.3.3. Data collection options

These options specify when events are to be aggregated into event counts. Also see [Limiting range of event collection](#).

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

Specify if you want Callgrind to start simulation and profiling from the beginning of the program. When set to `no`, Callgrind will not be able to collect any information, including calls, but it will have at most a slowdown of around 4, which is the minimum Valgrind overhead. Instrumentation can be interactively enabled via `callgrind_control -i on`.

Note that the resulting call graph will most probably not contain `main`, but will contain all the functions executed after instrumentation was enabled. Instrumentation can also be programmatically enabled/disabled. See the Callgrind include file `callgrind.h` for the macro you have to use in your source code.

For cache simulation, results will be less accurate when switching on instrumentation later in the program run, as the simulator starts with an empty cache at that moment. Switch on event collection later to cope with this error.

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

Specify whether event collection is enabled at beginning of the profile run.

To only look at parts of your program, you have two possibilities:

1. Zero event counters before entering the program part you want to profile, and dump the event counters to a file after leaving that program part.
2. Switch on/off collection state as needed to only see event counters happening while inside of the program part you want to profile.

The second option can be used if the program part you want to profile is called many times. Option 1, i.e. creating a lot of dumps is not practical here.

Collection state can be toggled at entry and exit of a given function with the option `--toggle-collect`. If you use this option, collection state should be disabled at the beginning. Note that the specification of `--toggle-collect` implicitly sets `--collect-state=no`.

Collection state can be toggled also by inserting the client request `CALLGRIND_TOGGLE_COLLECT` ; at the needed code positions.

`--toggle-collect=<function>`

Toggle collection on entry/exit of function.

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

This specifies whether information for (conditional) jumps should be collected. As above, `callgrind_annotate` currently is not able to show you the data. You have to use `KCachegrind` to get jump arrows in the annotated code.

`--collect-systime=<no|yes|msec|usec|nsec> [default: no]`

This specifies whether information for system call times should be collected.

The value `no` indicates to record no system call information.

The other values indicate to record the number of system calls done (`sysCount` event) and the elapsed time (`sysTime` event) spent in system calls. The `--collect-systime` value gives the unit used for `sysTime`: milli seconds, micro seconds or nano seconds. With the value `nsec`, `callgrind` also records the cpu time spent during system calls (`sysCpuTime`).

The value `yes` is a synonym of `msec`. The value `nsec` is not supported on Darwin.

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

This specifies whether the number of global bus events executed should be collected. The event type "Ge" is used for these events.

## 6.3.4. Cost entity separation options

These options specify how event counts should be attributed to execution contexts. For example, they specify whether the recursion level or the call chain leading to a function should be taken into account, and whether the thread ID should be considered. Also see [Avoiding cycles](#).

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

This option specifies whether profile data should be generated separately for every thread. If yes, the file names get "-threadID" appended.

`--separate-callers=<callers> [default: 0]`

Separate contexts by at most `<callers>` functions in the call chain. See [Avoiding cycles](#).

`--separate-callers<number>=<function>`

Separate number callers for function. See [Avoiding cycles](#).

`--separate-recs=<level> [default: 2]`

Separate function recursions by at most `level` levels. See [Avoiding cycles](#).

`--separate-recs<number>=<function>`

Separate number recursions for function. See [Avoiding cycles](#).

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

Ignore calls to/from PLT sections.

`--skip-direct-rec=<no|yes> [default: yes]`

Ignore direct recursions.

`--fn-skip=<function>`

Ignore calls to/from a given function. E.g. if you have a call chain `A > B > C`, and you specify function `B` to be ignored, you will only see `A > C`.

This is very convenient to skip functions handling callback behaviour. For example, with the signal/slot mechanism in the Qt graphics library, you only want to see the function emitting a signal to call the slots connected to that signal. First, determine the real call chain to see the functions needed to be skipped, then use this option.

### 6.3.5. Simulation options

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

Specify if you want to do full cache simulation. By default, only instruction read accesses will be counted ("Ir"). With cache simulation, further event counters are enabled: Cache misses on instruction reads ("I1mr"/"ILmr"), data read accesses ("Dr") and related cache misses ("D1mr"/"DLmr"), data write accesses ("Dw") and related cache misses ("D1mw"/"DLmw"). For more information, see [Cachegrind: a cache and branch-prediction profiler](#).

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

Specify if you want to do branch prediction simulation. Further event counters are enabled: Number of executed conditional branches and related predictor misses ("Bc"/"Bcm"), executed indirect jumps and related misses of the jump address predictor ("Bi"/"Bim").

### 6.3.6. Cache simulation options

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

Specify whether write-back behavior should be simulated, allowing to distinguish LL caches misses with and without write backs. The cache model of Cachegrind/Callgrind does not specify write-through vs. write-back behavior, and this also is not relevant for the number of generated miss counts. However, with explicit write-back simulation it can be decided whether a miss triggers not only the loading of a new cache line, but also if a write back of a dirty cache line had to take place before. The new dirty miss events are ILdmr, DLdmr, and DLdmw, for misses because of instruction read, data read, and data write, respectively. As they produce two memory transactions, they should account for a doubled time estimation in relation to a normal miss.

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

Specify whether simulation of a hardware prefetcher should be added which is able to detect stream access in the second level cache by comparing accesses to separate to each page. As the simulation can not decide about any timing issues of prefetching, it is assumed that any hardware prefetch triggered succeeds before a real access is done. Thus, this gives a best-case scenario by covering all possible stream accesses.

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

Specify whether cache line use should be collected. For every cache line, from loading to it being evicted, the number of accesses as well as the number of actually used bytes is determined. This behavior is related to the code which triggered loading of the cache line. In contrast to miss counters, which shows the position where the symptoms of bad cache behavior (i.e. latencies) happens, the use counters try to pinpoint at the reason (i.e. the code with the bad access behavior). The new counters are defined in a way such that worse behavior results in higher cost. AcCost1 and AcCost2 are counters showing bad temporal locality for L1 and LL caches, respectively. This is done by summing up reciprocal values of the numbers of accesses of each cache line, multiplied by 1000 (as only integer costs are allowed). E.g. for a given source line with 5 read accesses, a value of 5000 AcCost means that for every access, a new cache line was loaded and directly evicted afterwards without further accesses. Similarly, SpLoss1/2 shows bad spatial locality for L1 and LL caches, respectively. It gives the *spatial loss* count of bytes which were loaded into cache but never accessed. It pinpoints at code accessing data in a way such that cache space is wasted. This hints at bad layout of data structures in memory. Assuming a cache line size of 64 bytes and 100 L1 misses for a given source line, the loading of 6400 bytes into L1 was triggered. If SpLoss1 shows a value of 3200 for this line, this means that half of the loaded data was never used, or using a better data layout, only half of the cache space would have been needed. Please note that for cache line use counters, it currently is not possible to provide meaningful inclusive costs. Therefore, inclusive cost of these counters should be ignored.

`--I1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 instruction cache.

`--D1=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the level 1 data cache.

`--LL=<size>,<associativity>,<line size>`

Specify the size, associativity and line size of the last-level cache.

## 6.4. Callgrind Monitor Commands

The Callgrind tool provides monitor commands handled by the Valgrind gdbserver (see [Monitor command handling by the Valgrind gdbserver](#)). Valgrind python code provides GDB front end commands giving an easier usage of the callgrind monitor commands (see [GDB front end commands for Valgrind gdbserver monitor commands](#)). To launch a callgrind monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `callgrind` command (or the shorter aliases `cg`). Using the callgrind GDB front end command provide a more flexible usage, such as auto-completion of the command by GDB. In GDB, you can use `help callgrind` to get help about the callgrind front end monitor commands and you can use `apropos callgrind` to get all the commands mentioning the word "callgrind" in their name or on-line help.

- `dump` [`<dump_hint>`] requests to dump the profile data.
- `zero` requests to zero the profile data counters.
- `instrumentation` [`on|off`] requests to set (if parameter on/off is given) or get the current instrumentation state.
- `status` requests to print out some status information.

## 6.5. Callgrind specific client requests

Callgrind provides the following specific client requests in `callgrind.h`. See that file for the exact details of their arguments.

`CALLGRIND_DUMP_STATS`

Force generation of a profile dump at specified position in code, for the current thread only. Written counters will be reset to zero.

`CALLGRIND_DUMP_STATS_AT(string)`

Same as `CALLGRIND_DUMP_STATS`, but allows to specify a string to be able to distinguish profile dumps.

`CALLGRIND_ZERO_STATS`

Reset the profile counters for the current thread to zero.

`CALLGRIND_TOGGLE_COLLECT`

Toggle the collection state. This allows to ignore events with regard to profile counters. See also options `--collect-atstart` and `--toggle-collect`.

`CALLGRIND_START_INSTRUMENTATION`

Start full Callgrind instrumentation if not already enabled. When cache simulation is done, this will flush the simulated cache and lead to an artificial cache warmup phase afterwards with cache misses which would not have happened in reality. See also option `--instr-atstart`.

## CALLGRIND\_STOP\_INSTRUMENTATION

Stop full Callgrind instrumentation if not already disabled. This flushes Valgrinds translation cache, and does no additional instrumentation afterwards: it effectively will run at the same speed as Nulgrind, i.e. at minimal slowdown. Use this to speed up the Callgrind run for uninteresting code parts. Use [CALLGRIND\\_START\\_INSTRUMENTATION](#) to enable instrumentation again. See also option `--instr-atstart`.

## 6.6. callgrind\_annotate Command-line Options

`-h --help`

Show summary of options.

`--version`

Show version of callgrind\_annotate.

`--show=A,B,C [default: all]`

Only show figures for events A,B,C.

`--threshold=<0--100> [default: 99%]`

Percentage of counts (of primary sort event) we are interested in.

callgrind\_annotate stops printing functions when the sum of the cost percentage of the printed functions is bigger or equal to the given threshold percentage.

`--sort=A,B,C`

Sort columns by events A,B,C [event column order].

Optionally, each event is followed by a : and a threshold, to specify different thresholds depending on the event.

callgrind\_annotate stops printing functions when the sum of the cost percentage of the printed functions for all the events is bigger or equal to the given event threshold percentages.

When one or more thresholds are given via this option, the value of `--threshold` is ignored.

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

When enabled, a percentage is printed next to all event counts. This helps gauge the relative importance of each function and line.

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

Annotate all source files containing functions that helped reach the event count threshold.

`--context=N [default: 8]`

Print N lines of context before and after annotated lines.

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

Add subroutine costs to functions calls.

`--tree=<none|caller|calling|both> [default: none]`

Print for each function their callers, the called functions or both.

`-I, --include=<dir>`

Add `dir` to the list of directories to search for source files.

## 6.7. callgrind\_control Command-line Options

By default, `callgrind_control` acts on all programs run by the current user under Callgrind. It is possible to limit the actions to specified Callgrind runs by providing a list of pids or program names as argument. The default action is to give some brief information about the applications being run under Callgrind.

`-h --help`

Show a short description, usage, and summary of options.

`--version`

Show version of `callgrind_control`.

`-l --long`

Show also the working directory, in addition to the brief information given by default.

`-s --stat`

Show statistics information about active Callgrind runs.

`-b --back`

Show stack/back traces of each thread in active Callgrind runs. For each active function in the stack trace, also the number of invocations since program start (or last dump) is shown. This option can be combined with `-e` to show inclusive cost of active functions.

`-e [A,B,...] (default: all)`

Show the current per-thread, exclusive cost values of event counters. If no explicit event names are given, figures for all event types which are collected in the given Callgrind run are shown. Otherwise, only figures for event types A, B, ... are shown. If this option is combined with `-b`, inclusive cost for the functions of each active stack frame is provided, too.

`--dump[=<desc>] (default: no description)`

Request the dumping of profile information. Optionally, a description can be specified which is written into the dump as part of the information giving the reason which triggered the dump action. This can be used to distinguish multiple dumps.

`-z --zero`

Zero all event counters.

`-k --kill`

Force a Callgrind run to be terminated.

`--instr=<on|off>`

Switch instrumentation mode on or off. If a Callgrind run has instrumentation disabled, no simulation is done and no events are counted. This is useful to skip uninteresting program parts, as there is much less slowdown (same as with the Valgrind tool "none"). See also the Callgrind option `--instr-atstart`.

`--vgdb-prefix=<prefix>`

Specify the vgdb prefix to use by callgrind\_control. callgrind\_control internally uses vgdb to find and control the active Callgrind runs. If the `--vgdb-prefix` option was used for launching valgrind, then the same option must be given to callgrind\_control.

# 7. Helgrind: a thread error detector

To use this tool, you must specify `--tool=helgrind` on the Valgrind command line.

## 7.1. Overview

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.

The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Helgrind can detect three classes of errors, which are discussed in detail in the next three sections:

1. [Misuses of the POSIX pthreads API](#).
2. [Potential deadlocks arising from lock ordering problems](#).
3. [Data races -- accessing memory without adequate locking or synchronisation](#).

Problems like these often result in unreproducible, timing-dependent crashes, deadlocks and other misbehaviour, and can be difficult to find by other means.

Helgrind is aware of all the pthread abstractions and tracks their effects as accurately as it can. On x86 and amd64 platforms, it understands and partially handles implicit locking arising from the use of the LOCK instruction prefix. On PowerPC/POWER and ARM platforms, it partially handles implicit locking arising from load-linked and store-conditional instruction pairs.

Helgrind works best when your application uses only the POSIX pthreads API. However, if you want to use custom threading primitives, you can describe their behaviour to Helgrind using the `ANNOTATE_*` macros defined in `helgrind.h`.

Helgrind also provides [Execution Trees](#) memory profiling using the command line option `--xtree-memory` and the monitor command `xtmemory`.

Following those is a section containing [hints and tips on how to get the best out of Helgrind](#).

Then there is a [summary of command-line options](#).

Finally, there is [a brief summary of areas in which Helgrind could be improved](#).

## 7.2. Detected errors: Misuses of the POSIX pthreads API

Helgrind intercepts calls to many POSIX pthreads functions, and is therefore able to report on various common problems. Although these are unglamorous errors, their presence can lead to undefined program behaviour and hard-to-find bugs later on. The detected errors are:

- unlocking an invalid mutex
- unlocking a not-locked mutex
- unlocking a mutex held by a different thread
- destroying an invalid or a locked mutex
- recursively locking a non-recursive mutex

- deallocation of memory that contains a locked mutex
- passing mutex arguments to functions expecting reader-writer lock arguments, and vice versa
- when a POSIX pthread function fails with an error code that must be handled
- when a thread exits whilst still holding locked locks
- calling `pthread_cond_wait` with a not-locked mutex, an invalid mutex, or one locked by a different thread
- inconsistent bindings between condition variables and their associated mutexes
- invalid or duplicate initialisation of a pthread barrier
- initialisation of a pthread barrier on which threads are still waiting
- destruction of a pthread barrier object which was never initialised, or on which threads are still waiting
- waiting on an uninitialised pthread barrier
- for all of the pthreads functions that Helgrind intercepts, an error is reported, along with a stack trace, if the system threading library routine returns an error code, even if Helgrind itself detected no error

Checks pertaining to the validity of mutexes are generally also performed for reader-writer locks.

Various kinds of this-can't-possibly-happen events are also reported. These usually indicate bugs in the system threading library.

Reported errors always contain a primary stack trace indicating where the error was detected. They may also contain auxiliary stack traces giving additional information. In particular, most errors relating to mutexes will also tell you where that mutex first came to Helgrind's attention (the "was first observed at" part), so you have a chance of figuring out which mutex it is referring to. For example:

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
  at 0x4C2408D: pthread_mutex_unlock (hg_intercepts.c:492)
  by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
  by 0x40079B: main (tc09_bad_unlock.c:50)
Lock at 0x7FEFFFA90 was first observed
  at 0x4C25D01: pthread_mutex_init (hg_intercepts.c:326)
  by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
  by 0x40079B: main (tc09_bad_unlock.c:50)
```

Helgrind has a way of summarising thread identities, as you see here with the text "Thread #1". This is so that it can speak about threads and sets of threads without overwhelming you with details. See [below](#) for more information on interpreting error messages.

## 7.3. Detected errors: Inconsistent Lock Orderings

In this section, and in general, to "acquire" a lock simply means to lock that lock, and to "release" a lock means to unlock it.

Helgrind monitors the order in which threads acquire locks. This allows it to detect potential deadlocks which could arise from the formation of cycles of locks. Detecting such inconsistencies is useful because, whilst actual deadlocks are fairly obvious, potential deadlocks may never be discovered during testing and could later lead to hard-to-diagnose in-service failures.

The simplest example of such a problem is as follows.

- Imagine some shared resource R, which, for whatever reason, is guarded by two locks, L1 and L2, which must both be held when R is accessed.
- Suppose a thread acquires L1, then L2, and proceeds to access R. The implication of this is that all threads in the program must acquire the two locks in the order first L1 then L2. Not doing so risks deadlock.
- The deadlock could happen if two threads -- call them T1 and T2 -- both want to access R. Suppose T1 acquires L1 first, and T2 acquires L2 first. Then T1 tries to acquire L2, and T2 tries to acquire L1, but those locks are both already held. So T1 and T2 become deadlocked.

Helgrind builds a directed graph indicating the order in which locks have been acquired in the past. When a thread acquires a new lock, the graph is updated, and then checked to see if it now contains a cycle. The presence of a cycle indicates a potential deadlock involving the locks in the cycle.

In general, Helgrind will choose two locks involved in the cycle and show you how their acquisition ordering has become inconsistent. It does this by showing the program points that first defined the ordering, and the program points which later violated it. Here is a simple example involving just two locks:

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF0006A0" violated

Observed (incorrect) order is: acquisition of lock at 0x7FF0006A0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x400825: main (tc13_laogl.c:23)

followed by a later acquisition of lock at 0x7FF0006D0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x400853: main (tc13_laogl.c:24)

Required order was established by acquisition of lock at 0x7FF0006D0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x40076D: main (tc13_laogl.c:17)

followed by a later acquisition of lock at 0x7FF0006A0
  at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
  by 0x40079B: main (tc13_laogl.c:18)
```

When there are more than two locks in the cycle, the error is equally serious. However, at present Helgrind does not show the locks involved, sometimes because that information is not available, but also so as to avoid flooding you with information. For example, a naive implementation of the famous Dining Philosophers problem involves a cycle of five locks (see `helgrind/tests/tc14_laog_dinphils.c`). In this case Helgrind has detected that all 5 philosophers could simultaneously pick up their left fork and then deadlock whilst waiting to pick up their right forks.

```
Thread #6: lock order "0x80499A0 before 0x8049A00" violated

Observed (incorrect) order is: acquisition of lock at 0x8049A00
  at 0x40085BC: pthread_mutex_lock (hg_intercepts.c:495)
  by 0x80485B4: dine (tc14_laog_dinphils.c:18)
  by 0x400BDA4: mythread_wrapper (hg_intercepts.c:219)
  by 0x39B924: start_thread (pthread_create.c:297)
  by 0x2F107D: clone (clone.S:130)

followed by a later acquisition of lock at 0x80499A0
  at 0x40085BC: pthread_mutex_lock (hg_intercepts.c:495)
  by 0x80485CD: dine (tc14_laog_dinphils.c:19)
  by 0x400BDA4: mythread_wrapper (hg_intercepts.c:219)
  by 0x39B924: start_thread (pthread_create.c:297)
  by 0x2F107D: clone (clone.S:130)
```

## 7.4. Detected errors: Data Races

A data race happens, or could happen, when two threads access a shared memory location without using suitable locks or other synchronisation to ensure single-threaded access. Such missing locking can cause obscure timing dependent bugs. Ensuring programs are race-free is one of the central difficulties of threaded programming.

Reliably detecting races is a difficult problem, and most of Helgrind's internals are devoted to dealing with it. We begin with a simple example.

### 7.4.1. A Simple Data Race

About the simplest possible example of a race is as follows. In this program, it is impossible to know what the value of `var` is at the end of the program. Is it 2 ? Or 1 ?

```
#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
    var++; /* Unprotected relative to parent */ /* this is line 6 */
    return NULL;
}

int main ( void ) {
    pthread_t child;
    pthread_create(&child, NULL, child_fn, NULL);
    var++; /* Unprotected relative to child */ /* this is line 13 */
    pthread_join(child, NULL);
    return 0;
}
```

The problem is there is nothing to stop `var` being updated simultaneously by both threads. A correct program would protect `var` with a lock of type `pthread_mutex_t`, which is acquired before each access and released afterwards. Helgrind's output for this program is:

```
Thread #1 is the program's root thread

Thread #2 was created
  at 0x511C08E: clone (in /lib64/libc-2.8.so)
  by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
  by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
  by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
  by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
  at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
  at 0x4005DC: child_fn (simple_race.c:6)
  by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
  by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
  by 0x511C0CC: clone (in /lib64/libc-2.8.so)

Location 0x601038 is 0 bytes inside global var "var"
```

```
declared at simple_race.c:3
```

This is quite a lot of detail for an apparently simple error. The last clause is the main error message. It says there is a race as a result of a read of size 4 (bytes), at 0x601038, which is the address of `var`, happening in function `main` at line 13 in the program.

Two important parts of the message are:

- Helgrind shows two stack traces for the error, not one. By definition, a race involves two different threads accessing the same location in such a way that the result depends on the relative speeds of the two threads.

The first stack trace follows the text "Possible data race during read of size 4 ..." and the second trace follows the text "This conflicts with a previous write of size 4 ...". Helgrind is usually able to show both accesses involved in a race. At least one of these will be a write (since two concurrent, unsynchronised reads are harmless), and they will of course be from different threads.

By examining your program at the two locations, you should be able to get at least some idea of what the root cause of the problem is. For each location, Helgrind shows the set of locks held at the time of the access. This often makes it clear which thread, if any, failed to take a required lock. In this example neither thread holds a lock during the access.

- For races which occur on global or stack variables, Helgrind tries to identify the name and defining point of the variable. Hence the text "Location 0x601038 is 0 bytes inside global var "var" declared at simple\_race.c:3".

Showing names of stack and global variables carries no run-time overhead once Helgrind has your program up and running. However, it does require Helgrind to spend considerable extra time and memory at program startup to read the relevant debug info. Hence this facility is disabled by default. To enable it, you need to give the `--read-var-info=yes` option to Helgrind.

The following section explains Helgrind's race detection algorithm in more detail.

## 7.4.2. Helgrind's Race Detection Algorithm

Most programmers think about threaded programming in terms of the basic functionality provided by the threading library (POSIX Pthreads): thread creation, thread joining, locks, condition variables, semaphores and barriers.

The effect of using these functions is to impose constraints upon the order in which memory accesses can happen. This implied ordering is generally known as the "happens-before relation". Once you understand the happens-before relation, it is easy to see how Helgrind finds races in your code. Fortunately, the happens-before relation is itself easy to understand, and is by itself a useful tool for reasoning about the behaviour of parallel programs. We now introduce it using a simple example.

Consider first the following buggy program:

Parent thread:	Child thread:
<code>int var;</code>	
<code>// create child thread</code>	
<code>pthread_create(...)</code>	
<code>var = 20;</code>	<code>var = 10;</code>
	<code>exit</code>
<code>// wait for child</code>	
<code>pthread_join(...)</code>	
<code>printf("%d\n", var);</code>	

The parent thread creates a child. Both then write different values to some variable `var`, and the parent then waits for the child to exit.

What is the value of `var` at the end of the program, 10 or 20? We don't know. The program is considered buggy (it has a race) because the final value of `var` depends on the relative rates of progress of the parent and child threads. If the parent is fast and the child is slow, then the child's assignment may happen later, so the final value will be 10; and vice versa if the child is faster than the parent.

The relative rates of progress of parent vs child is not something the programmer can control, and will often change from run to run. It depends on factors such as the load on the machine, what else is running, the kernel's scheduling strategy, and many other factors.

The obvious fix is to use a lock to protect `var`. It is however instructive to consider a somewhat more abstract solution, which is to send a message from one thread to the other:

```

Parent thread:                                Child thread:

int var;

// create child thread
pthread_create(...)
var = 20;
// send message to child

// wait for child
pthread_join(...)
printf("%d\n", var);

                                // wait for message to arrive
                                var = 10;
                                exit

```

Now the program reliably prints "10", regardless of the speed of the threads. Why? Because the child's assignment cannot happen until after it receives the message. And the message is not sent until after the parent's assignment is done.

The message transmission creates a "happens-before" dependency between the two assignments: `var = 20;` must now happen-before `var = 10;`. And so there is no longer a race on `var`.

Note that it's not significant that the parent sends a message to the child. Sending a message from the child (after its assignment) to the parent (before its assignment) would also fix the problem, causing the program to reliably print "20".

Helgrind's algorithm is (conceptually) very simple. It monitors all accesses to memory locations. If a location -- in this example, `var`, is accessed by two different threads, Helgrind checks to see if the two accesses are ordered by the happens-before relation. If so, that's fine; if not, it reports a race.

It is important to understand that the happens-before relation creates only a partial ordering, not a total ordering. An example of a total ordering is comparison of numbers: for any two numbers  $x$  and  $y$ , either  $x$  is less than, equal to, or greater than  $y$ . A partial ordering is like a total ordering, but it can also express the concept that two elements are neither equal, less or greater, but merely unordered with respect to each other.

In the fixed example above, we say that `var = 20;` "happens-before" `var = 10;`. But in the original version, they are unordered: we cannot say that either happens-before the other.

What does it mean to say that two accesses from different threads are ordered by the happens-before relation? It means that there is some chain of inter-thread synchronisation operations which cause those accesses to happen in a particular order, irrespective of the actual rates of progress of the individual threads. This is a required property for a reliable threaded program, which is why Helgrind checks for it.

The happens-before relations created by standard threading primitives are as follows:

- When a mutex is unlocked by thread T1 and later (or immediately) locked by thread T2, then the memory accesses in T1 prior to the unlock must happen-before those in T2 after it acquires the lock.

- The same idea applies to reader-writer locks, although with some complication so as to allow correct handling of reads vs writes.
- When a condition variable (CV) is signalled on by thread T1 and some other thread T2 is thereby released from a wait on the same CV, then the memory accesses in T1 prior to the signalling must happen-before those in T2 after it returns from the wait. If no thread was waiting on the CV then there is no effect.
- If instead T1 broadcasts on a CV, then all of the waiting threads, rather than just one of them, acquire a happens-before dependency on the broadcasting thread at the point it did the broadcast.
- A thread T2 that continues after completing `sem_wait` on a semaphore that thread T1 posts on, acquires a happens-before dependence on the posting thread, a bit like dependencies caused mutex unlock-lock pairs. However, since a semaphore can be posted on many times, it is unspecified from which of the post calls the wait call gets its happens-before dependency.
- For a group of threads T1 .. Tn which arrive at a barrier and then move on, each thread after the call has a happens-after dependency from all threads before the barrier.
- A newly-created child thread acquires an initial happens-after dependency on the point where its parent created it. That is, all memory accesses performed by the parent prior to creating the child are regarded as happening-before all the accesses of the child.
- Similarly, when an exiting thread is reaped via a call to `pthread_join`, once the call returns, the reaping thread acquires a happens-after dependency relative to all memory accesses made by the exiting thread.

In summary: Helgrind intercepts the above listed events, and builds a directed acyclic graph represented the collective happens-before dependencies. It also monitors all memory accesses.

If a location is accessed by two different threads, but Helgrind cannot find any path through the happens-before graph from one access to the other, then it reports a race.

There are a couple of caveats:

- Helgrind doesn't check for a race in the case where both accesses are reads. That would be silly, since concurrent reads are harmless.
- Two accesses are considered to be ordered by the happens-before dependency even through arbitrarily long chains of synchronisation events. For example, if T1 accesses some location L, and then `pthread_cond_signals` T2, which later `pthread_cond_signals` T3, which then accesses L, then a suitable happens-before dependency exists between the first and second accesses, even though it involves two different inter-thread synchronisation events.

### 7.4.3. Interpreting Race Error Messages

Helgrind's race detection algorithm collects a lot of information, and tries to present it in a helpful way when a race is detected. Here's an example:

```
Thread #2 was created
  at 0x511C08E: clone (in /lib64/libc-2.8.so)
  by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
  by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
  by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
  by 0x4008F2: main (tc21_pthonce.c:86)

Thread #3 was created
  at 0x511C08E: clone (in /lib64/libc-2.8.so)
  by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
  by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
```

```
by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
by 0x4008F2: main (tc21_pthonce.c:86)
```

Possible data race during read of size 4 at 0x601070 by thread #3

Locks held: none

```
at 0x40087A: child (tc21_pthonce.c:74)
by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
by 0x511C0CC: clone (in /lib64/libc-2.8.so)
```

This conflicts with a previous write of size 4 by thread #2

Locks held: none

```
at 0x400883: child (tc21_pthonce.c:74)
by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
by 0x511C0CC: clone (in /lib64/libc-2.8.so)
```

Location 0x601070 is 0 bytes inside local var "unprotected2" declared at tc21\_pthonce.c:51, in frame #0 of thread 3

Helgrind first announces the creation points of any threads referenced in the error message. This is so it can speak concisely about threads without repeatedly printing their creation point call stacks. Each thread is only ever announced once, the first time it appears in any Helgrind error message.

The main error message begins at the text "Possible data race during read". At the start is information you would expect to see -- address and size of the racing access, whether a read or a write, and the call stack at the point it was detected.

A second call stack is presented starting at the text "This conflicts with a previous write". This shows a previous access which also accessed the stated address, and which is believed to be racing against the access in the first call stack. Note that this second call stack is limited to a maximum of `--history-backtrace-size` entries with a default value of 8 to limit the memory usage.

Finally, Helgrind may attempt to give a description of the raced-on address in source level terms. In this example, it identifies it as a local variable, shows its name, declaration point, and in which frame (of the first call stack) it lives. Note that this information is only shown when `--read-var-info=yes` is specified on the command line. That's because reading the DWARF3 debug information in enough detail to capture variable type and location information makes Helgrind much slower at startup, and also requires considerable amounts of memory, for large programs.

Once you have your two call stacks, how do you find the root cause of the race?

The first thing to do is examine the source locations referred to by each call stack. They should both show an access to the same location, or variable.

Now figure out how that location should have been made thread-safe:

- Perhaps the location was intended to be protected by a mutex? If so, you need to lock and unlock the mutex at both access points, even if one of the accesses is reported to be a read. Did you perhaps forget the locking at one or other of the accesses? To help you do this, Helgrind shows the set of locks held by each threads at the time they accessed the raced-on location.
- Alternatively, perhaps you intended to use some other scheme to make it safe, such as signalling on a condition variable. In all such cases, try to find a synchronisation event (or a chain thereof) which separates the earlier-observed access (as shown in the second call stack) from the later-observed access (as shown in the first call stack). In other words, try to find evidence that the earlier access "happens-before" the later access. See the previous subsection for an explanation of the happens-before relation.

The fact that Helgrind is reporting a race means it did not observe any happens-before relation between the two accesses. If Helgrind is working correctly, it should also be the case that you also cannot find any such relation,

even on detailed inspection of the source code. Hopefully, though, your inspection of the code will show where the missing synchronisation operation(s) should have been.

## 7.5. Hints and Tips for Effective Use of Helgrind

Helgrind can be very helpful in finding and resolving threading-related problems. Like all sophisticated tools, it is most effective when you understand how to play to its strengths.

Helgrind will be less effective when you merely throw an existing threaded program at it and try to make sense of any reported errors. It will be more effective if you design threaded programs from the start in a way that helps Helgrind verify correctness. The same is true for finding memory errors with Memcheck, but applies more here, because thread checking is a harder problem. Consequently it is much easier to write a correct program for which Helgrind falsely reports (threading) errors than it is to write a correct program for which Memcheck falsely reports (memory) errors.

With that in mind, here are some tips, listed most important first, for getting reliable results and avoiding false errors. The first two are critical. Any violations of them will swamp you with huge numbers of false data-race errors.

1. Make sure your application, and all the libraries it uses, use the POSIX threading primitives. Helgrind needs to be able to see all events pertaining to thread creation, exit, locking and other synchronisation events. To do so it intercepts many POSIX pthreads functions.

Do not roll your own threading primitives (mutexes, etc) from combinations of the Linux futex syscall, atomic counters, etc. These throw Helgrind's internal what's-going-on models way off course and will give bogus results.

Also, do not reimplement existing POSIX abstractions using other POSIX abstractions. For example, don't build your own semaphore routines or reader-writer locks from POSIX mutexes and condition variables. Instead use POSIX reader-writer locks and semaphores directly, since Helgrind supports them directly.

Helgrind directly supports the following POSIX threading abstractions: mutexes, reader-writer locks, condition variables (but see below), semaphores and barriers. Currently spinlocks are not supported, although they could be in future.

At the time of writing, the following popular Linux packages are known to implement their own threading primitives:

- Qt version 4.X. Qt 3.X is harmless in that it only uses POSIX pthreads primitives. Unfortunately Qt 4.X has its own implementation of mutexes (QMutex) and thread reaping. Helgrind 3.4.x contains direct support for Qt 4.X threading, which is experimental but is believed to work fairly well. A side effect of supporting Qt 4 directly is that Helgrind can be used to debug KDE4 applications. As this is an experimental feature, we would particularly appreciate feedback from folks who have used Helgrind to successfully debug Qt 4 and/or KDE4 applications.
- Runtime support library for GNU OpenMP (part of GCC), at least for GCC versions 4.2 and 4.3. The GNU OpenMP runtime library (`libgomp.so`) constructs its own synchronisation primitives using combinations of atomic memory instructions and the futex syscall, which causes total chaos since in Helgrind since it cannot "see" those.

Fortunately, this can be solved using a configuration-time option (for GCC). Rebuild GCC from source, and configure using `--disable-linux-futex`. This makes `libgomp.so` use the standard POSIX threading primitives instead. Note that this was tested using GCC 4.2.3 and has not been re-tested using more recent GCC versions. We would appreciate hearing about any successes or failures with more recent versions.

If you must implement your own threading primitives, there are a set of client request macros in `helgrind.h` to help you describe your primitives to Helgrind. You should be able to mark up mutexes, condition variables, etc, without difficulty.

It is also possible to mark up the effects of thread-safe reference counting using the `ANNOTATE_HAPPENS_BEFORE`, `ANNOTATE_HAPPENS_AFTER` and `ANNOTATE_HAPPENS_BEFORE_FORGET_ALL`, macros. Thread-safe reference counting using an atomically incremented/decremented refcount variable causes Helgrind problems because a one-to-zero transition of the reference count means the accessing thread has exclusive ownership of the associated resource (normally, a C++ object) and can therefore access it (normally, to run its destructor) without locking. Helgrind doesn't understand this, and markup is essential to avoid false positives.

Here are recommended guidelines for marking up thread safe reference counting in C++. You only need to mark up your release methods -- the ones which decrement the reference count. Given a class like this:

```
class MyClass {
    unsigned int mRefCount;

    void Release ( void ) {
        unsigned int newCount = atomic_decrement(&mRefCount);
        if (newCount == 0) {
            delete this;
        }
    }
}
```

the release method should be marked up as follows:

```
void Release ( void ) {
    unsigned int newCount = atomic_decrement(&mRefCount);
    if (newCount == 0) {
        ANNOTATE_HAPPENS_AFTER(&mRefCount);
        ANNOTATE_HAPPENS_BEFORE_FORGET_ALL(&mRefCount);
        delete this;
    } else {
        ANNOTATE_HAPPENS_BEFORE(&mRefCount);
    }
}
```

There are a number of complex, mostly-theoretical objections to this scheme. From a theoretical standpoint it appears to be impossible to devise a markup scheme which is completely correct in the sense of guaranteeing to remove all false races. The proposed scheme however works well in practice.

2. Avoid memory recycling. If you can't avoid it, you must use tell Helgrind what is going on via the `VALGRIND_HG_CLEAN_MEMORY` client request (in `helgrind.h`).

Helgrind is aware of standard heap memory allocation and deallocation that occurs via `malloc/free/new/delete` and from entry and exit of stack frames. In particular, when memory is deallocated via `free`, `delete`, or function exit, Helgrind considers that memory clean, so when it is eventually reallocated, its history is irrelevant.

However, it is common practice to implement memory recycling schemes. In these, memory to be freed is not handed to `free/delete`, but instead put into a pool of free buffers to be handed out again as required. The problem is that Helgrind has no way to know that such memory is logically no longer in use, and its history is irrelevant. Hence you must make that explicit, using the `VALGRIND_HG_CLEAN_MEMORY` client request to specify the relevant address ranges. It's easiest to put these requests into the pool manager code, and use them either when memory is returned to the pool, or is allocated from it.

3. Avoid POSIX condition variables. If you can, use POSIX semaphores (`sem_t`, `sem_post`, `sem_wait`) to do inter-thread event signalling. Semaphores with an initial value of zero are particularly useful for this.

Helgrind only partially correctly handles POSIX condition variables. This is because Helgrind can see inter-thread dependencies between a `pthread_cond_wait` call and a `pthread_cond_signal`/`pthread_cond_broadcast` call only if the waiting thread actually gets to the rendezvous first (so that it actually calls `pthread_cond_wait`). It can't see dependencies between the threads if the signaller arrives first. In the latter case, POSIX guidelines imply that the associated boolean condition still provides an inter-thread synchronisation event, but one which is invisible to Helgrind.

The result of Helgrind missing some inter-thread synchronisation events is to cause it to report false positives.

The root cause of this synchronisation lossage is particularly hard to understand, so an example is helpful. It was discussed at length by Arndt Muehlenfeld ("Runtime Race Detection in Multi-Threaded Programs", Dissertation, TU Graz, Austria). The canonical POSIX-recommended usage scheme for condition variables is as follows:

```

b   is a Boolean condition, which is False most of the time
cv  is a condition variable
mx  is its associated mutex

Signaller:                                Waiter:

lock(mx)                                  lock(mx)
b = True                                  while (b == False)
signal(cv)                                wait(cv,mx)
unlock(mx)                                unlock(mx)

```

Assume `b` is False most of the time. If the waiter arrives at the rendezvous first, it enters its while-loop, waits for the signaller to signal, and eventually proceeds. Helgrind sees the signal, notes the dependency, and all is well.

If the signaller arrives first, `b` is set to true, and the signal disappears into nowhere. When the waiter later arrives, it does not enter its while-loop and simply carries on. But even in this case, the waiter code following the while-loop cannot execute until the signaller sets `b` to True. Hence there is still the same inter-thread dependency, but this time it is through an arbitrary in-memory condition, and Helgrind cannot see it.

By comparison, Helgrind's detection of inter-thread dependencies caused by semaphore operations is believed to be exactly correct.

As far as I know, a solution to this problem that does not require source-level annotation of condition-variable wait loops is beyond the current state of the art.

4. Make sure you are using a supported Linux distribution. At present, Helgrind only properly supports glibc-2.3 or later. This in turn means we only support glibc's NPTL threading implementation. The old LinuxThreads implementation is not supported.
5. If your application is using thread local variables, helgrind might report false positive race conditions on these variables, despite being very probably race free. On Linux, you can use `--sim-hints=deactivate-pthread-stack-cache-via-hack` to avoid such false positive error messages (see [--sim-hints](#)).
6. Round up all finished threads using `pthread_join`. Avoid detaching threads: don't create threads in the detached state, and don't call `pthread_detach` on existing threads.

Using `pthread_join` to round up finished threads provides a clear synchronisation point that both Helgrind and programmers can see. If you don't call `pthread_join` on a thread, Helgrind has no way to know when it finishes, relative to any significant synchronisation points for other threads in the program. So it assumes that the thread lingers indefinitely and can potentially interfere indefinitely with the memory state of the program. It has every right to assume that -- after all, it might really be the case that, for scheduling reasons, the exiting thread did run very slowly in the last stages of its life.

7. Perform thread debugging (with Helgrind) and memory debugging (with Memcheck) together.

Helgrind tracks the state of memory in detail, and memory management bugs in the application are liable to cause confusion. In extreme cases, applications which do many invalid reads and writes (particularly to freed memory) have been known to crash Helgrind. So, ideally, you should make your application Memcheck-clean before using Helgrind.

It may be impossible to make your application Memcheck-clean unless you first remove threading bugs. In particular, it may be difficult to remove all reads and writes to freed memory in multithreaded C++ destructor sequences at program termination. So, ideally, you should make your application Helgrind-clean before using Memcheck.

Since this circularity is obviously unresolvable, at least bear in mind that Memcheck and Helgrind are to some extent complementary, and you may need to use them together.

8. POSIX requires that implementations of standard I/O (`printf`, `fprintf`, `fwrite`, `fread`, etc) are thread safe. Unfortunately GNU libc implements this by using internal locking primitives that Helgrind is unable to intercept. Consequently Helgrind generates many false race reports when you use these functions.

Helgrind attempts to hide these errors using the standard Valgrind error-suppression mechanism. So, at least for simple test cases, you don't see any. Nevertheless, some may slip through. Just something to be aware of.

9. Helgrind's error checks do not work properly inside the system threading library itself (`libpthread.so`), and it usually observes large numbers of (false) errors in there. Valgrind's suppression system then filters these out, so you should not see them.

If you see any race errors reported where `libpthread.so` or `ld.so` is the object associated with the innermost stack frame, please file a bug report at <http://www.valgrind.org/>.

## 7.6. Helgrind Command-line Options

The following end-user options are available:

```
--free-is-write=no|yes [default: no]
```

When enabled (not the default), Helgrind treats freeing of heap memory as if the memory was written immediately before the free. This exposes races where memory is referenced by one thread, and freed by another, but there is no observable synchronisation event to ensure that the reference happens before the free.

This functionality is new in Valgrind 3.7.0, and is regarded as experimental. It is not enabled by default because its interaction with custom memory allocators is not well understood at present. User feedback is welcomed.

```
--track-lockorders=no|yes [default: yes]
```

When enabled (the default), Helgrind performs lock order consistency checking. For some buggy programs, the large number of lock order errors reported can become annoying, particularly if you're only interested in race errors. You may therefore find it helpful to disable lock order checking.

```
--history-level=none|approx|full [default: full]
```

`--history-level=full` (the default) causes Helgrind to collect enough information about "old" accesses that it can produce two stack traces in a race report -- both the stack trace for the current access, and the trace for the older, conflicting access. To limit memory usage, "old" accesses stack traces are limited to a maximum of `--history-backtrace-size` entries (default 8) or to `--num-callers` value if this value is smaller.

Collecting such information is expensive in both speed and memory, particularly for programs that do many inter-thread synchronisation events (locks, unlocks, etc). Without such information, it is more difficult to track down the root causes of races. Nonetheless, you may not need it in situations where you just want to check for the presence or absence of races, for example, when doing regression testing of a previously race-free program.

`--history-level=none` is the opposite extreme. It causes Helgrind not to collect any information about previous accesses. This can be dramatically faster than `--history-level=full`.

`--history-level=approx` provides a compromise between these two extremes. It causes Helgrind to show a full trace for the later access, and approximate information regarding the earlier access. This approximate information consists of two stacks, and the earlier access is guaranteed to have occurred somewhere between program points denoted by the two stacks. This is not as useful as showing the exact stack for the previous access (as `--history-level=full` does), but it is better than nothing, and it is almost as fast as `--history-level=none`.

`--history-backtrace-size=<number>` [default: 8]

When `--history-level=full` is selected, `--history-backtrace-size=number` indicates how many entries to record in "old" accesses stack traces.

`--delta-stacktrace=no|yes` [default: yes on linux amd64/x86]

This flag only has any effect at `--history-level=full`.

`--delta-stacktrace` configures the way Helgrind captures the stacktraces for the option `--history-level=full`. Such a stacktrace is typically needed each time a new piece of memory is read or written in a basic block of instructions.

`--delta-stacktrace=no` causes Helgrind to compute a full history stacktrace from the unwind info each time a stacktrace is needed.

`--delta-stacktrace=yes` indicates to Helgrind to derive a new stacktrace from the previous stacktrace, as long as there was no call instruction, no return instruction, or any other instruction changing the call stack since the previous stacktrace was captured. If no such instruction was executed, the new stacktrace can be derived from the previous stacktrace by just changing the top frame to the current program counter. This option can speed up Helgrind by 25% when using `--history-level=full`.

The following aspects have to be considered when using `--delta-stacktrace=yes`:

- In some cases (for example in a function prologue), the valgrind unwinder might not properly unwind the stack, due to some limitations and/or due to wrong unwind info. When using `--delta-stacktrace=yes`, the wrong stack trace captured in the function prologue will be kept till the next call or return.
- On the other hand, `--delta-stacktrace=yes` sometimes helps to obtain a correct stacktrace, for example when the unwind info allows a correct stacktrace to be done in the beginning of the sequence, but not later on in the instruction sequence.
- Determining which instructions are changing the callstack is partially based on platform dependent heuristics, which have to be tuned/validated specifically for the platform. Also, unwinding in a function prologue must be good enough to allow using `--delta-stacktrace=yes`. Currently, the option `--delta-stacktrace=yes` has been reasonably validated only on linux x86 32 bits and linux amd64 64 bits. For more details about how to validate `--delta-stacktrace=yes`, see debug option `--hg-sanity-flags` and the function `check_cached_rcec_ok` in `libhb_core.c`.

`--conflict-cache-size=N` [default: 1000000]

This flag only has any effect at `--history-level=full`.

Information about "old" conflicting accesses is stored in a cache of limited size, with LRU-style management. This is necessary because it isn't practical to store a stack trace for every single memory access made by the program. Historical information on not recently accessed locations is periodically discarded, to free up space in the cache.

This option controls the size of the cache, in terms of the number of different memory addresses for which conflicting access information is stored. If you find that Helgrind is showing race errors with only one stack instead of the expected two stacks, try increasing this value.

The minimum value is 10,000 and the maximum is 30,000,000 (thirty times the default value). Increasing the value by 1 increases Helgrind's memory requirement by very roughly 100 bytes, so the maximum value will easily eat up three extra gigabytes or so of memory.

```
--check-stack-refs=no|yes [default: yes]
```

By default Helgrind checks all data memory accesses made by your program. This flag enables you to skip checking for accesses to thread stacks (local variables). This can improve performance, but comes at the cost of missing races on stack-allocated data.

```
--ignore-thread-creation=<yes|no> [default: no]
```

Controls whether all activities during thread creation should be ignored. By default enabled only on Solaris. Solaris provides higher throughput, parallelism and scalability than other operating systems, at the cost of more fine-grained locking activity. This means for example that when a thread is created under glibc, just one big lock is used for all thread setup. Solaris libc uses several fine-grained locks and the creator thread resumes its activities as soon as possible, leaving for example stack and TLS setup sequence to the created thread. This situation confuses Helgrind as it assumes there is some false ordering in place between creator and created thread; and therefore many types of race conditions in the application would not be reported. To prevent such false ordering, this command line option is set to `yes` by default on Solaris. All activity (loads, stores, client requests) is therefore ignored during:

- `pthread_create()` call in the creator thread
- thread creation phase (stack and TLS setup) in the created thread

Also new memory allocated during thread creation is untracked, that is race reporting is suppressed there. DRD does the same thing implicitly. This is necessary because Solaris libc caches many objects and reuses them for different threads and that confuses Helgrind.

## 7.7. Helgrind Monitor Commands

The Helgrind tool provides monitor commands handled by Valgrind's built-in gdbserver (see [Monitor command handling by the Valgrind gdbserver](#)). Valgrind python code provides GDB front end commands giving an easier usage of the helgrind monitor commands (see [GDB front end commands for Valgrind gdbserver monitor commands](#)). To launch an helgrind monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `helgrind` command (or the shorter aliases `hg`). Using the helgrind GDB front end command provide a more flexible usage, such as evaluation of address and length arguments by GDB. In GDB, you can use `help helgrind` to get help about the helgrind front end monitor commands and you can use `apropos helgrind` to get all the commands mentioning the word "helgrind" in their name or on-line help.

- `info locks [lock_addr]` shows the list of locks and their status. If `lock_addr` is given, only shows the lock located at this address.

In the following example, helgrind knows about one lock. This lock is located at the guest address `ga 0x8049a20`. The lock kind is `rdwr` indicating a reader-writer lock. Other possible lock kinds are `nonRec` (simple mutex, non recursive) and `mbRec` (simple mutex, possibly recursive). The lock kind is then followed by the list of threads holding the lock. In the below example, `R1:thread #6 tid 3` indicates that the helgrind thread #6 has acquired (once, as the counter following the letter R is one) the lock in read mode. The helgrind thread `nr` is incremented for each started thread. The presence of 'tid 3' indicates that the thread #6 is has not exited yet and is the valgrind tid 3. If a thread has terminated, then this is indicated with 'tid (exited)'.

```
(gdb) monitor info locks
Lock ga 0x8049a20 {
  kind    rdwr
  { R1:thread #6 tid 3 }
}
(gdb)
```

If you give the option `--read-var-info=yes`, then more information will be provided about the lock location, such as the global variable or the heap block that contains the lock:

```

Lock ga 0x8049a20 {
  Location 0x8049a20 is 0 bytes inside global var "s_rwlock"
  declared at rwlock_race.c:17
  kind    rdwr
  { R1:thread #3 tid 3 }
}

```

The GDB equivalent helgrind front end command `helgrind info locks [ADDR]` accept any address expression for its first ADDR argument.

- `accesshistory <addr> [<len>]` shows the access history recorded for <len> (default 1) bytes starting at <addr>. For each recorded access that overlaps with the given range, `accesshistory` shows the operation type (read or write), the address and size read or written, the helgrind thread nr/valgrind tid number that did the operation and the locks held by the thread at the time of the operation. The oldest access is shown first, the most recent access is shown last.

In the following example, we see first a recorded write of 4 bytes by thread #7 that has modified the given 2 bytes range. The second recorded write is the most recent recorded write : thread #9 modified the same 2 bytes as part of a 4 bytes write operation. The list of locks held by each thread at the time of the write operation are also shown.

```

(gdb) monitor accesshistory 0x8049D8A 2
write of size 4 at 0x8049D88 by thread #7 tid 3
==6319== Locks held: 2, at address 0x8049D8C (and 1 that can't be shown)
==6319==    at 0x804865F: child_fn1 (locked_vs_unlocked2.c:29)
==6319==    by 0x400AE61: mythread_wrapper (hg_intercepts.c:234)
==6319==    by 0x39B924: start_thread (pthread_create.c:297)
==6319==    by 0x2F107D: clone (clone.S:130)

write of size 4 at 0x8049D88 by thread #9 tid 2
==6319== Locks held: 2, at addresses 0x8049DA4 0x8049DD4
==6319==    at 0x804877B: child_fn2 (locked_vs_unlocked2.c:45)
==6319==    by 0x400AE61: mythread_wrapper (hg_intercepts.c:234)
==6319==    by 0x39B924: start_thread (pthread_create.c:297)
==6319==    by 0x2F107D: clone (clone.S:130)

```

The GDB equivalent helgrind front end command `helgrind accesshistory ADDR [LEN]` accept any address expression for its first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space, like in the following example:

```

(gdb) hg accesshistory &mx sizeof(mx)
read of size 4 at 0x1130A8 by thread #2 tid (exited)
==302== Locks held: none
==302==    at 0x1094AC: child8 (tc19_shadowmem.c:37)
==302==    by 0x10A0DF: steer (tc19_shadowmem.c:288)
==302==    by 0x48448A3: mythread_wrapper (hg_intercepts.c:406)
==302==    by 0x4879EA6: start_thread (pthread_create.c:477)
==302==    by 0x4990A2E: clone (clone.S:95)

```

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

## 7.8. Helgrind Client Requests

The following client requests are defined in `helgrind.h`. See that file for exact details of their arguments.

- `VALGRIND_HG_CLEAN_MEMORY`

This makes Helgrind forget everything it knows about a specified memory range. This is particularly useful for memory allocators that wish to recycle memory.

- `ANNOTATE_HAPPENS_BEFORE`
- `ANNOTATE_HAPPENS_AFTER`
- `ANNOTATE_NEW_MEMORY`
- `ANNOTATE_RWLOCK_CREATE`
- `ANNOTATE_RWLOCK_DESTROY`
- `ANNOTATE_RWLOCK_ACQUIRED`
- `ANNOTATE_RWLOCK_RELEASED`

These are used to describe to Helgrind, the behaviour of custom (non-POSIX) synchronisation primitives, which it otherwise has no way to understand. See comments in `helgrind.h` for further documentation.

## 7.9. A To-Do List for Helgrind

The following is a list of loose ends which should be tidied up some time.

- For lock order errors, print the complete lock cycle, rather than only doing for size-2 cycles as at present.
- The conflicting access mechanism sometimes mysteriously fails to show the conflicting access' stack, even when provided with unbounded storage for conflicting access info. This should be investigated.
- Document races caused by GCC's thread-unsafe code generation for speculative stores. In the interim see <http://gcc.gnu.org/ml/gcc/2007-10/msg00266.html> and <http://lkml.org/lkml/2007/10/24/673>.
- Don't update the lock-order graph, and don't check for errors, when a "try"-style lock operation happens (e.g. `pthread_mutex_trylock`). Such calls do not add any real restrictions to the locking order, since they can always fail to acquire the lock, resulting in the caller going off and doing Plan B (presumably it will have a Plan B). Doing such checks could generate false lock-order errors and confuse users.
- Performance can be very poor. Slowdowns on the order of 100:1 are not unusual. There is limited scope for performance improvements.

## 8. DRD: a thread error detector

To use this tool, you must specify `--tool=drd` on the Valgrind command line.

### 8.1. Overview

DRD is a Valgrind tool for detecting errors in multithreaded C and C++ programs. The tool works for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX threading primitives.

#### 8.1.1. Multithreaded Programming Paradigms

There are two possible reasons for using multithreading in a program:

- To model concurrent activities. Assigning one thread to each activity can be a great simplification compared to multiplexing the states of multiple activities in a single thread. This is why most server software and embedded software is multithreaded.
- To use multiple CPU cores simultaneously for speeding up computations. This is why many High Performance Computing (HPC) applications are multithreaded.

Multithreaded programs can use one or more of the following programming paradigms. Which paradigm is appropriate depends e.g. on the application type. Some examples of multithreaded programming paradigms are:

- Locking. Data that is shared over threads is protected from concurrent accesses via locking. E.g. the POSIX threads library, the Qt library and the Boost.Thread library support this paradigm directly.
- Message passing. No data is shared between threads, but threads exchange data by passing messages to each other. Examples of implementations of the message passing paradigm are MPI and CORBA.
- Automatic parallelization. A compiler converts a sequential program into a multithreaded program. The original program may or may not contain parallelization hints. One example of such parallelization hints is the OpenMP standard. In this standard a set of directives are defined which tell a compiler how to parallelize a C, C++ or Fortran program. OpenMP is well suited for computational intensive applications. As an example, an open source image processing software package is using OpenMP to maximize performance on systems with multiple CPU cores. GCC supports the OpenMP standard from version 4.2.0 on.
- Software Transactional Memory (STM). Any data that is shared between threads is updated via transactions. After each transaction it is verified whether there were any conflicting transactions. If there were conflicts, the transaction is aborted, otherwise it is committed. This is a so-called optimistic approach. There is a prototype of the Intel C++ Compiler available that supports STM. Research about the addition of STM support to GCC is ongoing.

DRD supports any combination of multithreaded programming paradigms as long as the implementation of these paradigms is based on the POSIX threads primitives. DRD however does not support programs that use e.g. Linux' `futexes` directly. Attempts to analyze such programs with DRD will cause DRD to report many false positives.

#### 8.1.2. POSIX Threads Programming Model

POSIX threads, also known as Pthreads, is the most widely available threading library on Unix systems.

The POSIX threads programming model is based on the following abstractions:

- A shared address space. All threads running within the same process share the same address space. All data, whether shared or not, is identified by its address.
- Regular load and store operations, which allow to read values from or to write values to the memory shared by all threads running in the same process.

- Atomic store and load-modify-store operations. While these are not mentioned in the POSIX threads standard, most microprocessors support atomic memory operations.
- Threads. Each thread represents a concurrent activity.
- Synchronization objects and operations on these synchronization objects. The following types of synchronization objects have been defined in the POSIX threads standard: mutexes, condition variables, semaphores, reader-writer synchronization objects, barriers and spinlocks.

Which source code statements generate which memory accesses depends on the *memory model* of the programming language being used. There is not yet a definitive memory model for the C and C++ languages. For a draft memory model, see also the document [WG21/N2338: Concurrency memory model compiler consequences](#).

For more information about POSIX threads, see also the Single UNIX Specification version 3, also known as [IEEE Std 1003.1](#).

### 8.1.3. Multithreaded Programming Problems

Depending on which multithreading paradigm is being used in a program, one or more of the following problems can occur:

- Data races. One or more threads access the same memory location without sufficient locking. Most but not all data races are programming errors and are the cause of subtle and hard-to-find bugs.
- Lock contention. One thread blocks the progress of one or more other threads by holding a lock too long.
- Improper use of the POSIX threads API. Most implementations of the POSIX threads API have been optimized for runtime speed. Such implementations will not complain on certain errors, e.g. when a mutex is being unlocked by another thread than the thread that obtained a lock on the mutex.
- Deadlock. A deadlock occurs when two or more threads wait for each other indefinitely.
- False sharing. If threads that run on different processor cores access different variables located in the same cache line frequently, this will slow down the involved threads a lot due to frequent exchange of cache lines.

Although the likelihood of the occurrence of data races can be reduced through a disciplined programming style, a tool for automatic detection of data races is a necessity when developing multithreaded software. DRD can detect these, as well as lock contention and improper use of the POSIX threads API.

### 8.1.4. Data Race Detection

The result of load and store operations performed by a multithreaded program depends on the order in which memory operations are performed. This order is determined by:

1. All memory operations performed by the same thread are performed in *program order*, that is, the order determined by the program source code and the results of previous load operations.
2. Synchronization operations determine certain ordering constraints on memory operations performed by different threads. These ordering constraints are called the *synchronization order*.

The combination of program order and synchronization order is called the *happens-before relationship*. This concept was first defined by S. Adve et al in the paper *Detecting data races on weak memory systems*, ACM SIGARCH Computer Architecture News, v.19 n.3, p.234-243, May 1991.

Two memory operations *conflict* if both operations are performed by different threads, refer to the same memory location and at least one of them is a store operation.

A multithreaded program is *data-race free* if all conflicting memory accesses are ordered by synchronization operations.

A well known way to ensure that a multithreaded program is data-race free is to ensure that a locking discipline is followed. It is e.g. possible to associate a mutex with each shared data item, and to hold a lock on the associated mutex while the shared data is accessed.

All programs that follow a locking discipline are data-race free, but not all data-race free programs follow a locking discipline. There exist multithreaded programs where access to shared data is arbitrated via condition variables, semaphores or barriers. As an example, a certain class of HPC applications consists of a sequence of computation steps separated in time by barriers, and where these barriers are the only means of synchronization. Although there are many conflicting memory accesses in such applications and although such applications do not make use of mutexes, most of these applications do not contain data races.

There exist two different approaches for verifying the correctness of multithreaded programs at runtime. The approach of the so-called Eraser algorithm is to verify whether all shared memory accesses follow a consistent locking strategy. And the happens-before data race detectors verify directly whether all interthread memory accesses are ordered by synchronization operations. While the last approach is more complex to implement, and while it is more sensitive to OS scheduling, it is a general approach that works for all classes of multithreaded programs. An important advantage of happens-before data race detectors is that these do not report any false positives.

DRD is based on the happens-before algorithm.

## 8.2. Using DRD

### 8.2.1. DRD Command-line Options

The following command-line options are available for controlling the behavior of the DRD tool itself:

`--check-stack-var=<yes|no> [default: no]`

Controls whether DRD detects data races on stack variables. Verifying stack variables is disabled by default because most programs do not share stack variables over threads.

`--exclusive-threshold=<n> [default: off]`

Print an error message if any mutex or writer lock has been held longer than the time specified in milliseconds. This option enables the detection of lock contention.

`--join-list-vol=<n> [default: 10]`

Data races that occur between a statement at the end of one thread and another thread can be missed if memory access information is discarded immediately after a thread has been joined. This option allows one to specify for how many joined threads memory access information should be retained.

`--first-race-only=<yes|no> [default: no]`

Whether to report only the first data race that has been detected on a memory location or all data races that have been detected on a memory location.

`--free-is-write=<yes|no> [default: no]`

Whether to report races between accessing memory and freeing memory. Enabling this option may cause DRD to run slightly slower. Notes:

- Don't enable this option when using custom memory allocators that use the `VG_USERREQ__MALLOCLIKE_BLOCK` and `VG_USERREQ__FREELIKE_BLOCK` because that would result in false positives.
- Don't enable this option when using reference-counted objects because that will result in false positives, even when that code has been annotated properly with `ANNOTATE_HAPPENS_BEFORE`

and `ANNOTATE_HAPPENS_AFTER`. See e.g. the output of the following command for an example: `valgrind --tool=drd --free-is-write=yes drd/tests/annotate_smart_pointer`.

`--report-signal-unlocked=<yes|no> [default: yes]`

Whether to report calls to `pthread_cond_signal` and `pthread_cond_broadcast` where the mutex associated with the signal through `pthread_cond_wait` or `pthread_cond_timed_wait` is not locked at the time the signal is sent. Sending a signal without holding a lock on the associated mutex is a common programming error which can cause subtle race conditions and unpredictable behavior. There exist some uncommon synchronization patterns however where it is safe to send a signal without holding a lock on the associated mutex.

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

Controls segment merging. Segment merging is an algorithm to limit memory usage of the data race detection algorithm. Disabling segment merging may improve the accuracy of the so-called 'other segments' displayed in race reports but can also trigger an out of memory error.

`--segment-merging-interval=<n> [default: 10]`

Perform segment merging only after the specified number of new segments have been created. This is an advanced configuration option that allows one to choose whether to minimize DRD's memory usage by choosing a low value or to let DRD run faster by choosing a slightly higher value. The optimal value for this parameter depends on the program being analyzed. The default value works well for most programs.

`--shared-threshold=<n> [default: off]`

Print an error message if a reader lock has been held longer than the specified time (in milliseconds). This option enables the detection of lock contention.

`--show-conf1-seg=<yes|no> [default: yes]`

Show conflicting segments in race reports. Since this information can help to find the cause of a data race, this option is enabled by default. Disabling this option makes the output of DRD more compact.

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

Print stack usage at thread exit time. When a program creates a large number of threads it becomes important to limit the amount of virtual memory allocated for thread stacks. This option makes it possible to observe how much stack memory has been used by each thread of the client program. Note: the DRD tool itself allocates some temporary data on the client thread stack. The space necessary for this temporary data must be allocated by the client program when it allocates stack memory, but is not included in stack usage reported by DRD.

`--ignore-thread-creation=<yes|no> [default: no]`

Controls whether all activities during thread creation should be ignored. By default enabled only on Solaris. Solaris provides higher throughput, parallelism and scalability than other operating systems, at the cost of more fine-grained locking activity. This means for example that when a thread is created under glibc, just one big lock is used for all thread setup. Solaris libc uses several fine-grained locks and the creator thread resumes its activities as soon as possible, leaving for example stack and TLS setup sequence to the created thread. This situation confuses DRD as it assumes there is some false ordering in place between creator and created thread; and therefore many types of race conditions in the application would not be reported. To prevent such false ordering, this command line option is set to `yes` by default on Solaris. All activity (loads, stores, client requests) is therefore ignored during:

- `pthread_create()` call in the creator thread
- thread creation phase (stack and TLS setup) in the created thread

The following options are available for monitoring the behavior of the client program:

```
--trace-addr=<address> [default: none]
```

Trace all load and store activity for the specified address. This option may be specified more than once.

```
--ptrace-addr=<address> [default: none]
```

Trace all load and store activity for the specified address and keep doing that even after the memory at that address has been freed and reallocated.

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

Trace all memory allocations and deallocations. May produce a huge amount of output.

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

Trace all barrier activity.

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

Trace all condition variable activity.

```
--trace-fork-join=<yes|no> [default: no]
```

Trace all thread creation and all thread termination events.

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

Trace execution of the `ANNOTATE_HAPPENS_BEFORE()`, `ANNOTATE_HAPPENS_AFTER()` and `ANNOTATE_HAPPENS_DONE()` client requests.

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

Trace all mutex activity.

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

Trace all reader-writer lock activity.

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

Trace all semaphore activity.

## 8.2.2. Detected Errors: Data Races

DRD prints a message every time it detects a data race. Please keep the following in mind when interpreting DRD's output:

- Every thread is assigned a *thread ID* by the DRD tool. A thread ID is a number. Thread ID's start at one and are never recycled.
- The term *segment* refers to a consecutive sequence of load, store and synchronization operations, all issued by the same thread. A segment always starts and ends at a synchronization operation. Data race analysis is performed between segments instead of between individual load and store operations because of performance reasons.
- There are always at least two memory accesses involved in a data race. Memory accesses involved in a data race are called *conflicting memory accesses*. DRD prints a report for each memory access that conflicts with a past memory access.

Below you can find an example of a message printed by DRD when it detects a data race:

```
$ valgrind --tool=drd --read-var-info=yes drd/tests/rwlock_race
...
```

```

==9466== Thread 3:
==9466== Conflicting load by thread 3 at 0x006020b8 size 4
==9466==   at 0x400B6C: thread_func (rwlock_race.c:29)
==9466==   by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==   by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Location 0x6020b8 is 0 bytes inside local var "s_racy"
==9466== declared at rwlock_race.c:18, in frame #0 of thread 3
==9466== Other segment start (thread 2)
==9466==   at 0x4C2847D: pthread_rwlock_rdlock* (drd_pthread_intercepts.c:813)
==9466==   by 0x400B6B: thread_func (rwlock_race.c:28)
==9466==   by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==   by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Other segment end (thread 2)
==9466==   at 0x4C28B54: pthread_rwlock_unlock* (drd_pthread_intercepts.c:912)
==9466==   by 0x400B84: thread_func (rwlock_race.c:30)
==9466==   by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==   by 0x53250CC: clone (in /lib64/libc-2.8.so)
...

```

The above report has the following meaning:

- The number in the column on the left is the process ID of the process being analyzed by DRD.
- The first line ("Thread 3") tells you the thread ID for the thread in which context the data race has been detected.
- The next line tells which kind of operation was performed (load or store) and by which thread. On the same line the start address and the number of bytes involved in the conflicting access are also displayed.
- Next, the call stack of the conflicting access is displayed. If your program has been compiled with debug information (`-g`), this call stack will include file names and line numbers. The two bottommost frames in this call stack (`clone` and `start_thread`) show how the NPTL starts a thread. The third frame (`vg_thread_wrapper`) is added by DRD. The fourth frame (`thread_func`) is the first interesting line because it shows the thread entry point, that is the function that has been passed as the third argument to `pthread_create`.
- Next, the allocation context for the conflicting address is displayed. For dynamically allocated data the allocation call stack is shown. For static variables and stack variables the allocation context is only shown when the option `--read-var-info=yes` has been specified. Otherwise DRD will print `Allocation context: unknown`.
- A conflicting access involves at least two memory accesses. For one of these accesses an exact call stack is displayed, and for the other accesses an approximate call stack is displayed, namely the start and the end of the segments of the other accesses. This information can be interpreted as follows:
  1. Start at the bottom of both call stacks, and count the number stack frames with identical function name, file name and line number. In the above example the three bottommost frames are identical (`clone`, `start_thread` and `vg_thread_wrapper`).
  2. The next higher stack frame in both call stacks now tells you between in which source code region the other memory access happened. The above output tells that the other memory access involved in the data race happened between source code lines 28 and 30 in file `rwlock_race.c`.

## 8.2.3. Detected Errors: Lock Contention

Threads must be able to make progress without being blocked for too long by other threads. Sometimes a thread has to wait until a mutex or reader-writer synchronization object is unlocked by another thread. This is called *lock contention*.

Lock contention causes delays. Such delays should be as short as possible. The two command line options `--exclusive-threshold=<n>` and `--shared-threshold=<n>` make it possible to detect excessive lock contention by making DRD report any lock that has been held longer than the specified threshold. An example:

```
$ valgrind --tool=drd --exclusive-threshold=10 drd/tests/hold_lock -i 500
...
==10668== Acquired at:
==10668==    at 0x4C267C8: pthread_mutex_lock (drd_pthread_intercepts.c:395)
==10668==    by 0x400D92: main (hold_lock.c:51)
==10668== Lock on mutex 0x7feffffd50 was held during 503 ms (threshold: 10 ms).
==10668==    at 0x4C26ADA: pthread_mutex_unlock (drd_pthread_intercepts.c:441)
==10668==    by 0x400DB5: main (hold_lock.c:55)
...
```

The `hold_lock` test program holds a lock as long as specified by the `-i` (interval) argument. The DRD output reports that the lock acquired at line 51 in source file `hold_lock.c` and released at line 55 was held during 503 ms, while a threshold of 10 ms was specified to DRD.

## 8.2.4. Detected Errors: Misuse of the POSIX threads API

DRD is able to detect and report the following misuses of the POSIX threads API:

- Passing the address of one type of synchronization object (e.g. a mutex) to a POSIX API call that expects a pointer to another type of synchronization object (e.g. a condition variable).
- Attempts to unlock a mutex that has not been locked.
- Attempts to unlock a mutex that was locked by another thread.
- Attempts to lock a mutex of type `PTHREAD_MUTEX_NORMAL` or a spinlock recursively.
- Destruction or deallocation of a locked mutex.
- Sending a signal to a condition variable while no lock is held on the mutex associated with the condition variable.
- Calling `pthread_cond_wait` on a mutex that is not locked, that is locked by another thread or that has been locked recursively.
- Associating two different mutexes with a condition variable through `pthread_cond_wait`.
- Destruction or deallocation of a condition variable that is being waited upon.
- Destruction or deallocation of a locked reader-writer synchronization object.
- Attempts to unlock a reader-writer synchronization object that was not locked by the calling thread.
- Attempts to recursively lock a reader-writer synchronization object exclusively.
- Attempts to pass the address of a user-defined reader-writer synchronization object to a POSIX threads function.
- Attempts to pass the address of a POSIX reader-writer synchronization object to one of the annotations for user-defined reader-writer synchronization objects.
- Reinitialization of a mutex, condition variable, reader-writer lock, semaphore or barrier.
- Destruction or deallocation of a semaphore or barrier that is being waited upon.
- Missing synchronization between barrier wait and barrier destruction.

- Exiting a thread without first unlocking the spinlocks, mutexes or reader-writer synchronization objects that were locked by that thread.
- Passing an invalid thread ID to `pthread_join` or `pthread_cancel`.

## 8.2.5. Client Requests

Just as for other Valgrind tools it is possible to let a client program interact with the DRD tool through client requests. In addition to the client requests several macros have been defined that allow to use the client requests in a convenient way.

The interface between client programs and the DRD tool is defined in the header file `<valgrind/drd.h>`. The available macros and client requests are:

- The macro `DRD_GET_VALGRIND_THREADID` and the corresponding client request `VG_USERREQ__DRD_GET_VALGRIND_THREAD_ID`. Query the thread ID that has been assigned by the Valgrind core to the thread executing this client request. Valgrind's thread ID's start at one and are recycled in case a thread stops.
- The macro `DRD_GET_DRD_THREADID` and the corresponding client request `VG_USERREQ__DRD_GET_DRD_THREAD_ID`. Query the thread ID that has been assigned by DRD to the thread executing this client request. These are the thread ID's reported by DRD in data race reports and in trace messages. DRD's thread ID's start at one and are never recycled.
- The macros `DRD_IGNORE_VAR(x)`, `ANNOTATE_TRACE_MEMORY(&x)` and the corresponding client request `VG_USERREQ__DRD_START_SUPPRESSION`. Some applications contain intentional races. There exist e.g. applications where the same value is assigned to a shared variable from two different threads. It may be more convenient to suppress such races than to solve these. This client request allows one to suppress such races.
- The macro `DRD_STOP_IGNOREING_VAR(x)` and the corresponding client request `VG_USERREQ__DRD_FINISH_SUPPRESSION`. Tell DRD to no longer ignore data races for the address range that was suppressed either via the macro `DRD_IGNORE_VAR(x)` or via the client request `VG_USERREQ__DRD_START_SUPPRESSION`.
- The macro `DRD_TRACE_VAR(x)`. Trace all load and store activity for the address range starting at `&x` and occupying `sizeof(x)` bytes. When DRD reports a data race on a specified variable, and it's not immediately clear which source code statements triggered the conflicting accesses, it can be very helpful to trace all activity on the offending memory location.
- The macro `DRD_STOP_TRACING_VAR(x)`. Stop tracing load and store activity for the address range starting at `&x` and occupying `sizeof(x)` bytes.
- The macro `ANNOTATE_TRACE_MEMORY(&x)`. Trace all load and store activity that touches at least the single byte at the address `&x`.
- The client request `VG_USERREQ__DRD_START_TRACE_ADDR`, which allows one to trace all load and store activity for the specified address range.
- The client request `VG_USERREQ__DRD_STOP_TRACE_ADDR`. Do no longer trace load and store activity for the specified address range.
- The macro `ANNOTATE_HAPPENS_BEFORE(addr)` tells DRD to insert a mark. Insert this macro just after an access to the variable at the specified address has been performed.
- The macro `ANNOTATE_HAPPENS_AFTER(addr)` tells DRD that the next access to the variable at the specified address should be considered to have happened after the access just before the latest `ANNOTATE_HAPPENS_BEFORE(addr)` annotation that references the same variable. The purpose of these two macros is to tell DRD about the order of inter-thread memory accesses implemented via atomic memory operations. See also `drd/tests/annotate_smart_pointer.cpp` for an example.

- The macro `ANNOTATE_RWLOCK_CREATE(rwlock)` tells DRD that the object at address `rwlock` is a reader-writer synchronization object that is not a `pthread_rwlock_t` synchronization object. See also `drd/tests/annotate_rwlock.c` for an example.
- The macro `ANNOTATE_RWLOCK_DESTROY(rwlock)` tells DRD that the reader-writer synchronization object at address `rwlock` has been destroyed.
- The macro `ANNOTATE_WRITERLOCK_ACQUIRED(rwlock)` tells DRD that a writer lock has been acquired on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_READERLOCK_ACQUIRED(rwlock)` tells DRD that a reader lock has been acquired on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_RWLOCK_ACQUIRED(rwlock, is_w)` tells DRD that a writer lock (when `is_w != 0`) or that a reader lock (when `is_w == 0`) has been acquired on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_WRITERLOCK_RELEASED(rwlock)` tells DRD that a writer lock has been released on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_READERLOCK_RELEASED(rwlock)` tells DRD that a reader lock has been released on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_RWLOCK_RELEASED(rwlock, is_w)` tells DRD that a writer lock (when `is_w != 0`) or that a reader lock (when `is_w == 0`) has been released on the reader-writer synchronization object at address `rwlock`.
- The macro `ANNOTATE_BARRIER_INIT(barrier, count, reinitialization_allowed)` tells DRD that a new barrier object at the address `barrier` has been initialized, that `count` threads participate in each barrier and also whether or not barrier reinitialization without intervening destruction should be reported as an error. See also `drd/tests/annotate_barrier.c` for an example.
- The macro `ANNOTATE_BARRIER_DESTROY(barrier)` tells DRD that a barrier object is about to be destroyed.
- The macro `ANNOTATE_BARRIER_WAIT_BEFORE(barrier)` tells DRD that waiting for a barrier will start.
- The macro `ANNOTATE_BARRIER_WAIT_AFTER(barrier)` tells DRD that waiting for a barrier has finished.
- The macro `ANNOTATE_BENIGN_RACE_SIZED(addr, size, descr)` tells DRD that any races detected on the specified address are benign and hence should not be reported. The `descr` argument is ignored but can be used to document why data races on `addr` are benign.
- The macro `ANNOTATE_BENIGN_RACE_STATIC(var, descr)` tells DRD that any races detected on the specified static variable are benign and hence should not be reported. The `descr` argument is ignored but can be used to document why data races on `var` are benign. Note: this macro can only be used in C++ programs and not in C programs.
- The macro `ANNOTATE_IGNORE_READS_BEGIN` tells DRD to ignore all memory loads performed by the current thread.
- The macro `ANNOTATE_IGNORE_READS_END` tells DRD to stop ignoring the memory loads performed by the current thread.
- The macro `ANNOTATE_IGNORE_WRITES_BEGIN` tells DRD to ignore all memory stores performed by the current thread.
- The macro `ANNOTATE_IGNORE_WRITES_END` tells DRD to stop ignoring the memory stores performed by the current thread.

- The macro `ANNOTATE_IGNORE_READS_AND_WRITES_BEGIN` tells DRD to ignore all memory accesses performed by the current thread.
- The macro `ANNOTATE_IGNORE_READS_AND_WRITES_END` tells DRD to stop ignoring the memory accesses performed by the current thread.
- The macro `ANNOTATE_NEW_MEMORY(addr, size)` tells DRD that the specified memory range has been allocated by a custom memory allocator in the client program and that the client program will start using this memory range.
- The macro `ANNOTATE_THREAD_NAME(name)` tells DRD to associate the specified name with the current thread and to include this name in the error messages printed by DRD.
- The macros `VALGRIND_MALLOCLIKE_BLOCK` and `VALGRIND_FREELIKE_BLOCK` from the Valgrind core are implemented; they are described in [The Client Request mechanism](#).

Note: if you compiled Valgrind yourself, the header file `<valgrind/drd.h>` will have been installed in the directory `/usr/include` by the command `make install`. If you obtained Valgrind by installing it as a package however, you will probably have to install another package with a name like `valgrind-devel` before Valgrind's header files are available.

## 8.2.6. Debugging C++11 Programs

If you want to use the C++11 class `std::thread` you will need to do the following to annotate the `std::shared_ptr<>` objects used in the implementation of that class:

- Add the following code at the start of a common header or at the start of each source file, before any C++ header files are included:

```
#include <valgrind/drd.h>
#define _GLIBCXX_SYNCHRONIZATION_HAPPENS_BEFORE(addr) ANNOTATE_HAPPENS_BEFORE(addr)
#define _GLIBCXX_SYNCHRONIZATION_HAPPENS_AFTER(addr) ANNOTATE_HAPPENS_AFTER(addr)
```

- Download the gcc source code and from source file `libstdc++-v3/src/c++11/thread.cc` copy the implementation of the `execute_native_thread_routine()` and `std::thread::_M_start_thread()` functions into a source file that is linked with your application. Make sure that also in this source file the `_GLIBCXX_SYNCHRONIZATION_HAPPENS_*` macros are defined properly.

For more information, see also *The GNU C++ Library Manual, Debugging Support* (<http://gcc.gnu.org/onlinedocs/libstdc++/manual/debug.html>).

## 8.2.7. Debugging GNOME Programs

GNOME applications use the threading primitives provided by the `glib` and `gthread` libraries. These libraries are built on top of POSIX threads, and hence are directly supported by DRD. Please keep in mind that you have to call `g_thread_init` before creating any threads, or DRD will report several data races on `glib` functions. See also the [GLib Reference Manual](#) for more information about `g_thread_init`.

One of the many facilities provided by the `glib` library is a block allocator, called `g_slice`. You have to disable this block allocator when using DRD by adding the following to the shell environment variables: `G_SLICE=always-malloc`. See also the [GLib Reference Manual](#) for more information.

## 8.2.8. Debugging Boost.Thread Programs

The `Boost.Thread` library is the threading library included with the cross-platform Boost Libraries. This threading library is an early implementation of the upcoming C++0x threading library.

Applications that use the `Boost.Thread` library should run fine under DRD.

More information about Boost.Thread can be found here:

- Anthony Williams, [Boost.Thread Library Documentation](#), Boost website, 2007.
- Anthony Williams, [What's New in Boost Threads?](#), Recent changes to the Boost Thread library, Dr. Dobbs Magazine, October 2008.

## 8.2.9. Debugging OpenMP Programs

OpenMP stands for *Open Multi-Processing*. The OpenMP standard consists of a set of compiler directives for C, C++ and Fortran programs that allows a compiler to transform a sequential program into a parallel program. OpenMP is well suited for HPC applications and allows one to work at a higher level compared to direct use of the POSIX threads API. While OpenMP ensures that the POSIX API is used correctly, OpenMP programs can still contain data races. So it definitely makes sense to verify OpenMP programs with a thread checking tool.

DRD supports OpenMP shared-memory programs generated by GCC. GCC supports OpenMP since version 4.2.0. GCC's runtime support for OpenMP programs is provided by a library called `libgomp`. The synchronization primitives implemented in this library use Linux' `futex` system call directly, unless the library has been configured with the `--disable-linux-futex` option. DRD only supports `libgomp` libraries that have been configured with this option and in which symbol information is present. For most Linux distributions this means that you will have to recompile GCC. See also the script `drd/scripts/download-and-build-gcc` in the Valgrind source tree for an example of how to compile GCC. You will also have to make sure that the newly compiled `libgomp.so` library is loaded when OpenMP programs are started. This is possible by adding a line similar to the following to your shell startup script:

```
export LD_LIBRARY_PATH=~/.gcc-4.4.0/lib64:~/.gcc-4.4.0/lib:
```

As an example, the test OpenMP test program `drd/tests/omp_matinv` triggers a data race when the option `-r` has been specified on the command line. The data race is triggered by the following code:

```
#pragma omp parallel for private(j)
for (j = 0; j < rows; j++)
{
    if (i != j)
    {
        const elem_t factor = a[j * cols + i];
        for (k = 0; k < cols; k++)
        {
            a[j * cols + k] -= a[i * cols + k] * factor;
        }
    }
}
```

The above code is racy because the variable `k` has not been declared private. DRD will print the following error message for the above code:

```
$ valgrind --tool=drd --check-stack-var=yes --read-var-info=yes drd/tests/omp_matinv 3
...
Conflicting store by thread 1/1 at 0x7feffffbc4 size 4
  at 0x4014A0: gj.omp_fn.0 (omp_matinv.c:203)
  by 0x401211: gj (omp_matinv.c:159)
  by 0x40166A: invert_matrix (omp_matinv.c:238)
  by 0x4019B4: main (omp_matinv.c:316)
Location 0x7feffffbc4 is 0 bytes inside local var "k"
declared at omp_matinv.c:160, in frame #0 of thread 1
...
```

In the above output the function name `g_j.omp_fn.0` has been generated by GCC from the function name `g_j`. The allocation context information shows that the data race has been caused by modifying the variable `k`.

Note: for GCC versions before 4.4.0, no allocation context information is shown. With these GCC versions the most usable information in the above output is the source file name and the line number where the data race has been detected (`omp_matinv.c:203`).

For more information about OpenMP, see also [openmp.org](http://openmp.org).

## 8.2.10. DRD and Custom Memory Allocators

DRD tracks all memory allocation events that happen via the standard memory allocation and deallocation functions (`malloc`, `free`, `new` and `delete`), via entry and exit of stack frames or that have been annotated with Valgrind's memory pool client requests. DRD uses memory allocation and deallocation information for two purposes:

- To know where the scope ends of POSIX objects that have not been destroyed explicitly. It is e.g. not required by the POSIX threads standard to call `pthread_mutex_destroy` before freeing the memory in which a mutex object resides.
- To know where the scope of variables ends. If e.g. heap memory has been used by one thread, that thread frees that memory, and another thread allocates and starts using that memory, no data races must be reported for that memory.

It is essential for correct operation of DRD that the tool knows about memory allocation and deallocation events. When analyzing a client program with DRD that uses a custom memory allocator, either instrument the custom memory allocator with the `VALGRIND_MALLOCLIKE_BLOCK` and `VALGRIND_FREELIKE_BLOCK` macros or disable the custom memory allocator.

As an example, the GNU `libstdc++` library can be configured to use standard memory allocation functions instead of memory pools by setting the environment variable `GLIBCXX_FORCE_NEW`. For more information, see also the [libstdc++ manual](#).

## 8.2.11. DRD Versus Memcheck

It is essential for correct operation of DRD that there are no memory errors such as dangling pointers in the client program. Which means that it is a good idea to make sure that your program is Memcheck-clean before you analyze it with DRD. It is possible however that some of the Memcheck reports are caused by data races. In this case it makes sense to run DRD before Memcheck.

So which tool should be run first? In case both DRD and Memcheck complain about a program, a possible approach is to run both tools alternately and to fix as many errors as possible after each run of each tool until none of the two tools prints any more error messages.

## 8.2.12. Resource Requirements

The requirements of DRD with regard to heap and stack memory and the effect on the execution time of client programs are as follows:

- When running a program under DRD with default DRD options, between 1.1 and 3.6 times more memory will be needed compared to a native run of the client program. More memory will be needed if loading debug information has been enabled (`--read-var-info=yes`).
- DRD allocates some of its temporary data structures on the stack of the client program threads. This amount of data is limited to 1 - 2 KB. Make sure that thread stacks are sufficiently large.
- Most applications will run between 20 and 50 times slower under DRD than a native single-threaded run. The slowdown will be most noticeable for applications which perform frequent mutex lock / unlock operations.

## 8.2.13. Hints and Tips for Effective Use of DRD

The following information may be helpful when using DRD:

- Make sure that debug information is present in the executable being analyzed, such that DRD can print function name and line number information in stack traces. Most compilers can be told to include debug information via compiler option `-g`.
- Compile with option `-O1` instead of `-O0`. This will reduce the amount of generated code, may reduce the amount of debug info and will speed up DRD's processing of the client program. For more information, see also [Getting started](#).
- If DRD reports any errors on libraries that are part of your Linux distribution like e.g. `libc.so` or `libstdc++.so`, installing the debug packages for these libraries will make the output of DRD a lot more detailed.
- When using C++, do not send output from more than one thread to `std::cout`. Doing so would not only generate multiple data race reports, it could also result in output from several threads getting mixed up. Either use `printf` or do the following:
  1. Derive a class from `std::ostreambuf` and let that class send output line by line to `stdout`. This will avoid that individual lines of text produced by different threads get mixed up.
  2. Create one instance of `std::ostream` for each thread. This makes stream formatting settings thread-local. Pass a per-thread instance of the class derived from `std::ostreambuf` to the constructor of each instance.
  3. Let each thread send its output to its own instance of `std::ostream` instead of `std::cout`.

## 8.3. Using the POSIX Threads API Effectively

### 8.3.1. Mutex types

The Single UNIX Specification version two defines the following four mutex types (see also the documentation of [pthread\\_mutexattr\\_t](#)):

- *normal*, which means that no error checking is performed, and that the mutex is non-recursive.
- *error checking*, which means that the mutex is non-recursive and that error checking is performed.
- *recursive*, which means that a mutex may be locked recursively.
- *default*, which means that error checking behavior is undefined, and that the behavior for recursive locking is also undefined. Or: portable code must neither trigger error conditions through the Pthreads API nor attempt to lock a mutex of default type recursively.

In complex applications it is not always clear from beforehand which mutex will be locked recursively and which mutex will not be locked recursively. Attempts lock a non-recursive mutex recursively will result in race conditions that are very hard to find without a thread checking tool. So either use the error checking mutex type and consistently check the return value of Pthread API mutex calls, or use the recursive mutex type.

### 8.3.2. Condition variables

A condition variable allows one thread to wake up one or more other threads. Condition variables are often used to notify one or more threads about state changes of shared data. Unfortunately it is very easy to introduce race conditions by using condition variables as the only means of state information propagation. A better approach is to let threads poll for changes of a state variable that is protected by a mutex, and to use condition variables only as a thread wakeup mechanism. See also the source file `drd/tests/monitor_example.cpp` for an example of how to implement this concept in C++. The monitor concept used in this example is a well known and very useful concept -- see also Wikipedia for more information about the [monitor](#) concept.

### 8.3.3. pthread\_cond\_timedwait and timeouts

Historically the function `pthread_cond_timedwait` only allowed the specification of an absolute timeout, that is a timeout independent of the time when this function was called. However, almost every call to this function expresses a relative timeout. This typically happens by passing the sum of `clock_gettime(CLOCK_REALTIME)` and a relative timeout as the third argument. This approach is incorrect since forward or backward clock adjustments by e.g. ntpd will affect the timeout. A more reliable approach is as follows:

- When initializing a condition variable through `pthread_cond_init`, specify that the timeout of `pthread_cond_timedwait` will use the clock `CLOCK_MONOTONIC` instead of `CLOCK_REALTIME`. You can do this via `pthread_condattr_setclock(..., CLOCK_MONOTONIC)`.
- When calling `pthread_cond_timedwait`, pass the sum of `clock_gettime(CLOCK_MONOTONIC)` and a relative timeout as the third argument.

See also `drd/tests/monitor_example.cpp` for an example.

## 8.4. Limitations

DRD currently has the following limitations:

- DRD, just like Memcheck, will refuse to start on Linux distributions where all symbol information has been removed from `ld.so`. This is e.g. the case for the PPC editions of openSUSE and Gentoo. You will have to install the `glibc debuginfo` package on these platforms before you can use DRD. See also openSUSE bug [396197](#) and Gentoo bug [214065](#).
- With gcc 4.4.3 and before, DRD may report data races on the C++ class `std::string` in a multithreaded program. This is a known `libstdc++` issue -- see also GCC bug [40518](#) for more information.
- If you compile the DRD source code yourself, you need GCC 3.0 or later. GCC 2.95 is not supported.
- Of the two POSIX threads implementations for Linux, only the NPTL (Native POSIX Thread Library) is supported. The older LinuxThreads library is not supported.

## 8.5. Feedback

If you have any comments, suggestions, feedback or bug reports about DRD, feel free to either post a message on the Valgrind users mailing list or to file a bug report. See also <http://www.valgrind.org/> for more information.

## 9. Massif: a heap profiler

To use this tool, you must specify `--tool=massif` on the Valgrind command line.

### 9.1. Overview

Massif is a heap profiler. It measures how much heap memory your program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of your program's stack(s), although it does not do so by default.

Heap profiling can help you reduce the amount of memory your program uses. On modern machines with virtual memory, this provides the following benefits:

- It can speed up your program -- a smaller program will interact better with your machine's caches and avoid paging.
- If your program uses lots of memory, it will reduce the chance that it exhausts your machine's swap space.

Also, there are certain space leaks that aren't detected by traditional leak-checkers, such as Memcheck's. That's because the memory isn't ever actually lost -- a pointer remains to it -- but it's not in use. Programs that have leaks like this can unnecessarily increase the amount of memory they are using over time. Massif can help identify these leaks.

Importantly, Massif tells you not only how much heap memory your program is using, it also gives very detailed information that indicates which parts of your program are responsible for allocating the heap memory.

Massif also provides [Execution Trees](#) memory profiling using the command line option `--xtree-memory` and the monitor command `xtmemory`.

### 9.2. Using Massif and ms\_print

First off, as for the other Valgrind tools, you should compile with debugging info (the `-g` option). It shouldn't matter much what optimisation level you compile your program with, as this is unlikely to affect the heap memory usage.

Then, you need to run Massif itself to gather the profiling information, and then run `ms_print` to present it in a readable way.

#### 9.2.1. An Example Program

An example will make things clear. Consider the following C program (annotated with line numbers) which allocates a number of different blocks on the heap.

```
1      #include <stdlib.h>
2
3      void g(void)
4      {
5          malloc(4000);
6      }
7
8      void f(void)
9      {
10         malloc(2000);
11         g();
12     }
13
14     int main(void)
```

```

15     {
16         int i;
17         int* a[10];
18
19         for (i = 0; i < 10; i++) {
20             a[i] = malloc(1000);
21         }
22
23         f();
24
25         g();
26
27         for (i = 0; i < 10; i++) {
28             free(a[i]);
29         }
30
31         return 0;
32     }

```

## 9.2.2. Running Massif

To gather heap profiling information about the program `prog`, type:

```
valgrind --tool=massif prog
```

The program will execute (slowly). Upon completion, no summary statistics are printed to Valgrind's commentary; all of Massif's profiling data is written to a file. By default, this file is called `massif.out.<pid>`, where `<pid>` is the process ID, although this filename can be changed with the `--massif-out-file` option.

## 9.2.3. Running ms\_print

To see the information gathered by Massif in an easy-to-read form, use `ms_print`. If the output file's name is `massif.out.12345`, type:

```
ms_print massif.out.12345
```

`ms_print` will produce (a) a graph showing the memory consumption over the program's execution, and (b) detailed information about the responsible allocation sites at various points in the program, including the point of peak memory allocation. The use of a separate script for presenting the results is deliberate: it separates the data gathering from its presentation, and means that new methods of presenting the data can be added in the future.

## 9.2.4. The Output Preamble

After running this program under Massif, the first part of `ms_print`'s output contains a preamble which just states how the program, Massif and `ms_print` were each invoked:

```

-----
Command:                example
Massif arguments:        (none)
ms_print arguments:      massif.out.12797
-----

```

## 9.2.5. The Output Graph

The next part is the graph that shows how memory consumption occurred as the program executed:



Why is most of the graph empty, with only a couple of bars at the very end? By default, Massif uses "instructions executed" as the unit of time. For very short-run programs such as the example, most of the executed instructions involve the loading and dynamic linking of the program. The execution of `main` (and thus the heap allocations) only occur at the very end. For a short-running program like this, we can use the `--time-unit=B` option to specify that we want the time unit to instead be the number of bytes allocated/deallocated on the heap and stack(s).

If we re-run the program under Massif with this option, and then re-run `ms_print`, we get this more useful graph:



```
Number of snapshots: 25
Detailed snapshots: [9, 14 (peak), 24]
```

The size of the graph can be changed with `ms_print's --x` and `--y` options. Each vertical bar represents a snapshot, i.e. a measurement of the memory usage at a certain point in time. If the next snapshot is more than one column away, a horizontal line of characters is drawn from the top of the snapshot to just before the next snapshot column. The text at the bottom show that 25 snapshots were taken for this program, which is one per heap allocation/deallocation, plus a couple of extras. Massif starts by taking snapshots for every heap allocation/deallocation, but as a program runs for longer, it takes snapshots less frequently. It also discards older snapshots as the program goes on; when it reaches the maximum number of snapshots (100 by default, although changeable with the `--max-snapshots` option) half of them are deleted. This means that a reasonable number of snapshots are always maintained.

Most snapshots are *normal*, and only basic information is recorded for them. Normal snapshots are represented in the graph by bars consisting of `'.'` characters.

Some snapshots are *detailed*. Information about where allocations happened are recorded for these snapshots, as we will see shortly. Detailed snapshots are represented in the graph by bars consisting of `'@'` characters. The text at the bottom show that 3 detailed snapshots were taken for this program (snapshots 9, 14 and 24). By default, every 10th snapshot is detailed, although this can be changed via the `--detailed-freq` option.

Finally, there is at most one *peak* snapshot. The peak snapshot is a detailed snapshot, and records the point where memory consumption was greatest. The peak snapshot is represented in the graph by a bar consisting of `'#'` characters. The text at the bottom shows that snapshot 14 was the peak.

Massif's determination of when the peak occurred can be wrong, for two reasons.

- Peak snapshots are only ever taken after a deallocation happens. This avoids lots of unnecessary peak snapshot recordings (imagine what happens if your program allocates a lot of heap blocks in succession, hitting a new peak every time). But it means that if your program never deallocates any blocks, no peak will be recorded. It also means that if your program does deallocate blocks but later allocates to a higher peak without subsequently deallocating, the reported peak will be too low.
- Even with this behaviour, recording the peak accurately is slow. So by default Massif records a peak whose size is within 1% of the size of the true peak. This inaccuracy in the peak measurement can be changed with the `--peak-inaccuracy` option.

The following graph is from an execution of Konqueror, the KDE web browser. It shows what graphs for larger programs look like.





Number of snapshots: 63

Detailed snapshots: [3, 4, 10, 11, 15, 16, 29, 33, 34, 36, 39, 41, 42, 43, 44, 49, 50, 51, 53, 55, 56, 57 (peak)]

Note that the larger size units are KB, MB, GB, etc. As is typical for memory measurements, these are based on a multiplier of 1024, rather than the standard SI multiplier of 1000. Strictly speaking, they should be written KiB, MiB, GiB, etc.

## 9.2.6. The Snapshot Details

Returning to our example, the graph is followed by the detailed information for each snapshot. The first nine snapshots are normal, so only a small amount of information is recorded for each one:

n	time(B)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	1,008	1,008	1,000	8	0
2	2,016	2,016	2,000	16	0
3	3,024	3,024	3,000	24	0
4	4,032	4,032	4,000	32	0
5	5,040	5,040	5,000	40	0
6	6,048	6,048	6,000	48	0
7	7,056	7,056	7,000	56	0
8	8,064	8,064	8,000	64	0

Each normal snapshot records several things.

- Its number.
- The time it was taken. In this case, the time unit is bytes, due to the use of `--time-unit=B`.
- The total memory consumption at that point.
- The number of useful heap bytes allocated at that point. This reflects the number of bytes asked for by the program.
- The number of extra heap bytes allocated at that point. This reflects the number of bytes allocated in excess of what the program asked for. There are two sources of extra heap bytes.

First, every heap block has administrative bytes associated with it. The exact number of administrative bytes depends on the details of the allocator. By default Massif assumes 8 bytes per block, as can be seen from the example, but this number can be changed via the `--heap-admin` option.

Second, allocators often round up the number of bytes asked for to a larger number, usually 8 or 16. This is required to ensure that elements within the block are suitably aligned. If *N* bytes are asked for, Massif rounds *N* up to the nearest multiple of the value specified by the `--alignment` option.

- The size of the stack(s). By default, stack profiling is off as it slows Massif down greatly. Therefore, the stack column is zero in the example. Stack profiling can be turned on with the `--stacks=yes` option.

The next snapshot is detailed. As well as the basic counts, it gives an allocation tree which indicates exactly which pieces of code were responsible for allocating heap memory:

```

 9          9,072          9,072          9,000          72          0
99.21% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.21% (9,000B) 0x804841A: main (example.c:20)

```

The allocation tree can be read from the top down. The first line indicates all heap allocation functions such as `malloc` and C++ `new`. All heap allocations go through these functions, and so all 9,000 useful bytes (which is 99.21% of all allocated bytes) go through them. But how were `malloc` and `new` called? At this point, every allocation so far has been due to line 20 inside `main`, hence the second line in the tree. The `->` indicates that `main` (line 20) called `malloc`.

Let's see what the subsequent output shows happened next:

```

-----
  n          time(B)          total(B)    useful-heap(B)  extra-heap(B)    stacks(B)
-----
 10          10,080          10,080          10,000          80              0
 11          12,088          12,088          12,000          88              0
 12          16,096          16,096          16,000          96              0
 13          20,104          20,104          20,000          104             0
 14          20,104          20,104          20,000          104             0
99.48% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.74% (10,000B) 0x804841A: main (example.c:20)
|
->39.79% (8,000B) 0x80483C2: g (example.c:5)
| ->19.90% (4,000B) 0x80483E2: f (example.c:11)
| | ->19.90% (4,000B) 0x8048431: main (example.c:23)
| |
| ->19.90% (4,000B) 0x8048436: main (example.c:25)
|
->09.95% (2,000B) 0x80483DA: f (example.c:10)
  ->09.95% (2,000B) 0x8048431: main (example.c:23)

```

The first four snapshots are similar to the previous ones. But then the global allocation peak is reached, and a detailed snapshot (number 14) is taken. Its allocation tree shows that 20,000B of useful heap memory has been allocated, and the lines and arrows indicate that this is from three different code locations: line 20, which is responsible for 10,000B (49.74%); line 5, which is responsible for 8,000B (39.79%); and line 10, which is responsible for 2,000B (9.95%).

We can then drill down further in the allocation tree. For example, of the 8,000B asked for by line 5, half of it was due to a call from line 11, and half was due to a call from line 25.

In short, Massif collates the stack trace of every single allocation point in the program into a single tree, which gives a complete picture at a particular point in time of how and why all heap memory was allocated.

Note that the tree entries correspond not to functions, but to individual code locations. For example, if function A calls `malloc`, and function B calls A twice, once on line 10 and once on line 11, then the two calls will result in two distinct stack traces in the tree. In contrast, if B calls A repeatedly from line 15 (e.g. due to a loop), then each of those calls will be represented by the same stack trace in the tree.

Note also that each tree entry with children in the example satisfies an invariant: the entry's size is equal to the sum of its children's sizes. For example, the first entry has size 20,000B, and its children have sizes 10,000B, 8,000B, and 2,000B. In general, this invariant almost always holds. However, in rare circumstances stack traces can be malformed, in which case a stack trace can be a sub-trace of another stack trace. This means that some entries in the tree may not satisfy the invariant -- the entry's size will be greater than the sum of its children's sizes. This is not a big problem, but could make the results confusing. Massif can sometimes detect when this happens; if it does, it issues a warning:

```
Warning: Malformed stack trace detected. In Massif's output,
```

the size of an entry's child entries may not sum up to the entry's size as they normally do.

However, Massif does not detect and warn about every such occurrence. Fortunately, malformed stack traces are rare in practice.

Returning now to `ms_print`'s output, the final part is similar:

n	time(B)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
15	21,112	19,096	19,000	96	0
16	22,120	18,088	18,000	88	0
17	23,128	17,080	17,000	80	0
18	24,136	16,072	16,000	72	0
19	25,144	15,064	15,000	64	0
20	26,152	14,056	14,000	56	0
21	27,160	13,048	13,000	48	0
22	28,168	12,040	12,000	40	0
23	29,176	11,032	11,000	32	0
24	30,184	10,024	10,000	24	0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.					
->79.81% (8,000B) 0x80483C2: g (example.c:5)					
->39.90% (4,000B) 0x80483E2: f (example.c:11)					
->39.90% (4,000B) 0x8048431: main (example.c:23)					
->39.90% (4,000B) 0x8048436: main (example.c:25)					
->19.95% (2,000B) 0x80483DA: f (example.c:10)					
->19.95% (2,000B) 0x8048431: main (example.c:23)					
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)					

The final detailed snapshot shows how the heap looked at termination. The 00.00% entry represents the code locations for which memory was allocated and then freed (line 20 in this case, the memory for which was freed on line 28). However, no code location details are given for this entry; by default, Massif only records the details for code locations responsible for more than 1% of useful memory bytes, and `ms_print` likewise only prints the details for code locations responsible for more than 1%. The entries that do not meet this threshold are aggregated. This avoids filling up the output with large numbers of unimportant entries. The thresholds can be changed with the `--threshold` option that both Massif and `ms_print` support.

## 9.2.7. Forking Programs

If your program forks, the child will inherit all the profiling data that has been gathered for the parent.

If the output file format string (controlled by `--massif-out-file`) does not contain `%p`, then the outputs from the parent and child will be intermingled in a single output file, which will almost certainly make it unreadable by `ms_print`.

## 9.2.8. Measuring All Memory in a Process

It is worth emphasising that by default Massif measures only heap memory, i.e. memory allocated with `malloc`, `calloc`, `realloc`, `memalign`, `new`, `new[]`, and a few other, similar functions. (And it can optionally measure stack memory, of course.) This means it does *not* directly measure memory allocated with lower-level system calls such as `mmap`, `mremap`, and `brk`.

Heap allocation functions such as `malloc` are built on top of these system calls. For example, when needed, an allocator will typically call `mmap` to allocate a large chunk of memory, and then hand over pieces of that memory

chunk to the client program in response to calls to `malloc` et al. Massif directly measures only these higher-level `malloc` et al calls, not the lower-level system calls.

Furthermore, a client program may use these lower-level system calls directly to allocate memory. By default, Massif does not measure these. Nor does it measure the size of code, data and BSS segments. Therefore, the numbers reported by Massif may be significantly smaller than those reported by tools such as `top` that measure a program's total size in memory.

However, if you wish to measure *all* the memory used by your program, you can use the `--pages-as-heap=yes`. When this option is enabled, Massif's normal heap block profiling is replaced by lower-level page profiling. Every page allocated via `mmap` and similar system calls is treated as a distinct block. This means that code, data and BSS segments are all measured, as they are just memory pages. Even the stack is measured, since it is ultimately allocated (and extended when necessary) via `mmap`; for this reason `--stacks=yes` is not allowed in conjunction with `--pages-as-heap=yes`.

After `--pages-as-heap=yes` is used, `ms_print`'s output is mostly unchanged. One difference is that the start of each detailed snapshot says:

```
(page allocation syscalls) mmap/mremap/brk, --alloc-fns, etc.
```

instead of the usual:

```
(heap allocation functions) malloc/new/new[], --alloc-fns, etc.
```

The stack traces in the output may be more difficult to read, and interpreting them may require some detailed understanding of the lower levels of a program like the memory allocators. But for some programs having the full information about memory usage can be very useful.

## 9.2.9. Acting on Massif's Information

Massif's information is generally fairly easy to act upon. The obvious place to start looking is the peak snapshot.

It can also be useful to look at the overall shape of the graph, to see if memory usage climbs and falls as you expect; spikes in the graph might be worth investigating.

The detailed snapshots can get quite large. It is worth viewing them in a very wide window. It's also a good idea to view them with a text editor. That makes it easy to scroll up and down while keeping the cursor in a particular column, which makes following the allocation chains easier.

## 9.3. Using massif-visualizer

[massif-visualizer](#) is a graphical viewer for Massif data that is often easier to use than `ms_print`. `massif-visualizer` is not shipped within Valgrind, but is available in various places online.

## 9.4. Massif Command-line Options

Massif-specific command-line options are:

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

Specifies whether heap profiling should be done.

```
--heap-admin=<size> [default: 8]
```

If heap profiling is enabled, gives the number of administrative bytes per block to use. This should be an estimate of the average, since it may vary. For example, the allocator used by `glibc` on Linux requires

somewhere between 4 to 15 bytes per block, depending on various factors. That allocator also requires admin space for freed blocks, but Massif cannot account for this.

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

Specifies whether stack profiling should be done. This option slows Massif down greatly, and so is off by default. Note that Massif assumes that the main stack has size zero at start-up. This is not true, but doing otherwise accurately is difficult. Furthermore, starting at zero better indicates the size of the part of the main stack that a user program actually has control over.

`--pages-as-heap=<yes|no> [default: no]`

Tells Massif to profile memory at the page level rather than at the malloc'd block level. See above for details.

`--depth=<number> [default: 30]`

Maximum depth of the allocation trees recorded for detailed snapshots. Increasing it will make Massif run somewhat more slowly, use more memory, and produce bigger output files.

`--alloc-fn=<name>`

Functions specified with this option will be treated as though they were a heap allocation function such as `malloc`. This is useful for functions that are wrappers to `malloc` or `new`, which can fill up the allocation trees with uninteresting information. This option can be specified multiple times on the command line, to name multiple functions.

Note that the named function will only be treated this way if it is the top entry in a stack trace, or just below another function treated this way. For example, if you have a function `malloc1` that wraps `malloc`, and `malloc2` that wraps `malloc1`, just specifying `--alloc-fn=malloc2` will have no effect. You need to specify `--alloc-fn=malloc1` as well. This is a little inconvenient, but the reason is that checking for allocation functions is slow, and it saves a lot of time if Massif can stop looking through the stack trace entries as soon as it finds one that doesn't match rather than having to continue through all the entries.

Note that C++ names are demangled. Note also that overloaded C++ names must be written in full. Single quotes may be necessary to prevent the shell from breaking them up. For example:

```
--alloc-fn='operator new(unsigned, std::nothrow_t const&)'
```

Arguments of type `size_t` need to be replaced with `unsigned long` on 64bit platforms and `unsigned` on 32bit platforms.

`--alloc-fn` will work with inline functions. Inline function names are not mangled, which means that you only need to provide the function name and not the argument list.

`--alloc-fn` does not support wildcards.

`--ignore-fn=<name>`

Any direct heap allocation (i.e. a call to `malloc`, `new`, etc, or a call to a function named by an `--alloc-fn` option) that occurs in a function specified by this option will be ignored. This is mostly useful for testing purposes. This option can be specified multiple times on the command line, to name multiple functions.

Any `realloc` of an ignored block will also be ignored, even if the `realloc` call does not occur in an ignored function. This avoids the possibility of negative heap sizes if ignored blocks are shrunk with `realloc`.

The rules for writing C++ function names are the same as for `--alloc-fn` above.

`--threshold=<m.n> [default: 1.0]`

The significance threshold for heap allocations, as a percentage of total memory size. Allocation tree entries that account for less than this will be aggregated. Note that this should be specified in tandem with `ms_print`'s option of the same name.

`--peak-inaccuracy=<m.n> [default: 1.0]`

Massif does not necessarily record the actual global memory allocation peak; by default it records a peak only when the global memory allocation size exceeds the previous peak by at least 1.0%. This is because there can be many local allocation peaks along the way, and doing a detailed snapshot for every one would be expensive and wasteful, as all but one of them will be later discarded. This inaccuracy can be changed (even to 0.0%) via this option, but Massif will run drastically slower as the number approaches zero.

`--time-unit=<i|ms|B> [default: i]`

The time unit used for the profiling. There are three possibilities: instructions executed (i), which is good for most cases; real (wallclock) time (ms, i.e. milliseconds), which is sometimes useful; and bytes allocated/deallocated on the heap and/or stack (B), which is useful for very short-run programs, and for testing purposes, because it is the most reproducible across different machines.

`--detailed-freq=<n> [default: 10]`

Frequency of detailed snapshots. With `--detailed-freq=1`, every snapshot is detailed.

`--max-snapshots=<n> [default: 100]`

The maximum number of snapshots recorded. If set to N, for all programs except very short-running ones, the final number of snapshots will be between N/2 and N.

`--massif-out-file=<file> [default: massif.out.%p]`

Write the profile data to `file` rather than to the default output file, `massif.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

## 9.5. Massif Monitor Commands

The Massif tool provides monitor commands handled by the Valgrind gdbserver (see [Monitor command handling by the Valgrind gdbserver](#)). Valgrind python code provides GDB front end commands giving an easier usage of the massif monitor commands (see [GDB front end commands for Valgrind gdbserver monitor commands](#)). To launch a massif monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `massif` command (or the shorter aliases `ms`). Using the massif GDB front end command provide a more flexible usage, such as auto-completion of the command by GDB. In GDB, you can use `help massif` to get help about the massif front end monitor commands and you can use `apropos massif` to get all the commands mentioning the word "massif" in their name or on-line help.

- `snapshot [<filename>]` requests to take a snapshot and save it in the given `<filename>` (default `massif.vgdb.out`).
- `detailed_snapshot [<filename>]` requests to take a detailed snapshot and save it in the given `<filename>` (default `massif.vgdb.out`).
- `all_snapshots [<filename>]` requests to take all captured snapshots so far and save them in the given `<filename>` (default `massif.vgdb.out`).
- `xtmemory [<filename> default xtmemory.kcg.%p.%n]` requests Massif tool to produce an xtree heap memory report. See [Execution Trees](#) for a detailed explanation about execution trees.

## 9.6. Massif Client Requests

Massif does not have a `massif.h` file, but it does implement two of the core client requests: `VALGRIND_MALLOCLIKE_BLOCK` and `VALGRIND_FREELIKE_BLOCK`; they are described in [The Client Request mechanism](#).

## 9.7. ms\_print Command-line Options

ms\_print's options are:

`-h --help`

Show the help message.

`--version`

Show the version number.

`--threshold=<m.n> [default: 1.0]`

Same as Massif's `--threshold` option, but applied after profiling rather than during.

`--x=<4..1000> [default: 72]`

Width of the graph, in columns.

`--y=<4..1000> [default: 20]`

Height of the graph, in rows.

## 9.8. Massif's Output File Format

Massif's file format is plain text (i.e. not binary) and deliberately easy to read for both humans and machines. Nonetheless, the exact format is not described here. This is because the format is currently very Massif-specific. In the future we hope to make the format more general, and thus suitable for possible use with other tools. Once this has been done, the format will be documented here.

# 10. DHAT: a dynamic heap analysis tool

To use this tool, you must specify `--tool=dhat` on the Valgrind command line.

## 10.1. Overview

DHAT is primarily a tool for examining how programs use their heap allocations.

It tracks the allocated blocks, and inspects every memory access to find which block, if any, it is to. It presents, on a program point basis, information about these blocks such as sizes, lifetimes, numbers of reads and writes, and read and write patterns.

Using this information it is possible to identify program points with the following characteristics:

- potential process-lifetime leaks: blocks allocated by the point just accumulate, and are freed only at the end of the run.
- excessive turnover: points which chew through a lot of heap, even if it is not held onto for very long
- excessively transient: points which allocate very short lived blocks
- useless or underused allocations: blocks which are allocated but not completely filled in, or are filled in but not subsequently read.
- blocks with inefficient layout -- areas never accessed, or with hot fields scattered throughout the block.

As with the Massif heap profiler, DHAT measures program progress by counting instructions, and so presents all age/time related figures as instruction counts. This sounds a little odd at first, but it makes runs repeatable in a way which is not possible if CPU time is used.

DHAT also has support for copy profiling and ad hoc profiling. These are described below.

## 10.2. Using DHAT

First off, as for normal Valgrind use, you probably want to compile with debugging info (the `-g` option). But by contrast with normal Valgrind use, you probably do want to turn optimisation on, since you should profile your program as it will be normally run.

Second, you need to run your program under DHAT to gather the profiling information. You might need to reduce the `--num-callers` value to get reasonably-sized output files, especially if you are profiling a large program; some trial and error might be needed to find a good value.

Finally, you need to use DHAT's viewer (in a web browser) to get a detailed presentation of that information.

### 10.2.1. Running DHAT

To run DHAT on a program `prog`, run:

```
valgrind --tool=dhat prog
```

The program will execute (slowly). Upon completion, summary statistics that look like this will be printed:

```

==11514== Total:      823,849,731 bytes in 3,929,133 blocks
==11514== At t-gmax: 133,485,082 bytes in 436,521 blocks
==11514== At t-end:   258,002 bytes in 2,129 blocks
==11514== Reads:      2,807,182,810 bytes
==11514== Writes:     1,149,617,086 bytes

```

The first line shows how many heap blocks and bytes were allocated over the entire execution.

The second line shows how many heap blocks and bytes were alive at `t-gmax`, i.e. the time when the heap size reached its global maximum (as measured in bytes).

The third line shows how many heap blocks and bytes were alive at `t-end`, i.e. the end of execution. In other words, how many blocks and bytes were not explicitly freed.

The fourth and fifth lines show how many bytes within heap blocks were read and written during the entire execution.

These lines are moderately interesting at best. More useful information can be seen with DHAT's viewer.

## 10.2.2. Output File

As well as printing summary information, DHAT also writes more detailed profiling information to a file. By default this file is named `dhat.out.<pid>` (where `<pid>` is the program's process ID), but its name can be changed with the `--dhat-out-file` option. This file is JSON, and intended to be viewed by DHAT's viewer, which is described in the next section.

The default `.<pid>` suffix on the output file name serves two purposes. Firstly, it means you don't have to rename old log files that you don't want to overwrite. Secondly, and more importantly, it allows correct profiling with the `--trace-children=yes` option of programs that spawn child processes.

The output file can be big, many megabytes for large applications built with full debugging information.

## 10.3. DHAT's Viewer

DHAT's viewer can be run in a web browser by loading the file `dh_view.html`. Use the "Load" button to choose a DHAT output file to view.

If loading takes a long time, it might be worth re-running DHAT with a smaller `--num-callers` value to reduce the stack depths, because this can significantly reduce the size of DHAT's output files.

### 10.3.1. The Output Header

The first part of the output shows the mode, program command and process ID. For example:

```

Invocation {
  Mode:      heap
  Command:   /home/njn/moz/rust0/build/x86_64-unknown-linux-gnu/stage2/bin/rustc --crate-
  PID:       18816
}

```

The second part of the output shows the `t-gmax` and `t-end` values again. For example:

```

Times {
  t-gmax: 8,138,210,673 instrs (86.92% of program duration)
  t-end:  9,362,544,994 instrs
}

```

## 10.3.2. The PP Tree

The third part of the output is the largest and most interesting part, showing the program point (PP) tree.

### 10.3.2.1. Structure

The following image shows a screenshot of part of a PP tree. The font is very small because this screenshot is intended to demonstrate the high-level structure of the tree rather than the details within the text. (It is also slightly out-of-date, and doesn't quite match the current output produced by DHAT's viewer.)

Like any tree, it has a root node, leaf nodes, and non-leaf nodes. The structure of the tree is shown by the lines connecting nodes. Child nodes are beneath their parent and indented one level.

The sub-trees beneath a non-leaf node can be collapsed or expanded by clicking on the node. It is useful to collapse sub-trees that you aren't interested in.

Colours are meaningful, and are intended to ease tree navigation, but the information they represent is also present within the text. (This means that colour-blind users are not denied any information.)

Each leaf node is coloured green. Each non-leaf node is coloured blue and has a down arrow (#) next to it when its sub-tree is expanded. Each non-leaf node is coloured yellow and has a left arrow (#) next to it when its sub-tree is collapsed.

The shade of green, blue or yellow used for a node indicate its significance. Darker shades represent greater significance (in terms of bytes or blocks).

Note that the entire output is text, even the arrows and lines connecting nodes. This means you can copy and paste any part of the output easily into an email, bug report, etc.

### 10.3.2.2. The Root Node

The root node looks like this:

```
PP 1/1 (25 children) {
  Total:      1,355,253,987 bytes (100%, 67,454.81/Minstr) in 5,943,417 blocks (100%, 29
  At t-gmax: 423,930,307 bytes (100%) in 1,575,682 blocks (100%), avg size 269.05 bytes
  At t-end:   258,002 bytes (100%) in 2,129 blocks (100%), avg size 121.18 bytes
  Reads:      5,478,606,988 bytes (100%, 272,685.7/Minstr), 4.04/byte
  Writes:     2,040,294,800 bytes (100%, 101,551.22/Minstr), 1.51/byte
  Allocated at {
    #0: [root]
  }
}
```

The root node covers the entire execution. The information is a superset of the information shown when DHAT ran, adding details such as allocation rates, average block sizes, block lifetimes, and read and write ratios. The next example will explain these in more detail.

### 10.3.2.3. Interior Nodes

PP nodes further down the tree show information about a subset of allocations. For example:

```
PP 1.1/25 (2 children) {
  Total:      54,533,440 bytes (4.02%, 2,714.28/Minstr) in 458,839 blocks (7.72%, 22.84/
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:   0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
```

```

Reads:      15,993,012 bytes (0.29%, 796.02/Minstr), 0.29/byte
Writes:     20,974,752 bytes (1.03%, 1,043.97/Minstr), 0.38/byte
Allocated at {
  #1: 0x95CACC9: alloc (alloc.rs:72)
  #2: 0x95CACC9: alloc (alloc.rs:148)
  #3: 0x95CACC9: reserve_internal<syntax::tokenstream::TokenStream,alloc::alloc::Global> (raw_
  #4: 0x95CACC9: reserve<syntax::tokenstream::TokenStream,alloc::alloc::Global> (raw_
  #5: 0x95CACC9: reserve<syntax::tokenstream::TokenStream> (vec.rs:460)
  #6: 0x95CACC9: push<syntax::tokenstream::TokenStream> (vec.rs:989)
  #7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
  #8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::String
}
}

```

The first line indicates the node's position in the tree. The 1.1 is a unique identifier for the node and also says that it is the first child node 1 (which is the root). The /25 says that it is one of 25 children, i.e. it has 24 siblings. The (2 children) says that this node node has two children of its own.

Allocations are aggregated by their allocation stack trace. The `Allocated at` section shows the allocation stack trace that is shared by all the blocks covered by this node.

The `Total` line shows that this node accounts for 4.02% of all bytes allocated during execution, and 7.72% of all blocks. These percentages are useful for comparing the significance of different nodes within a single profile; a PP that accounts for 10% of bytes allocated is likely to be more interesting than one that accounts for 2%.

The `Total` line also shows allocation rates, measured in bytes and blocks per million instructions. These rates are useful for comparing the significance of nodes across profiles made with different workloads.

Finally, the `Total` line shows the average size and lifetimes of these blocks.

The `At t-gmax` line says shows that no blocks from this PP were alive when the global heap peak occurred. In other words, these blocks do not contribute at all to the global heap peak.

The `At t-end` line shows that no blocks were from this PP were alive at shutdown. In other words, all those blocks were explicitly freed before termination.

The `Reads` and `Writes` lines show how many bytes were read within this PP's blocks, the fraction this represents of all heap reads, and the read rate. Finally, it shows the read ratio, which is the number of reads per byte. In this case the number is 0.29, which is quite low -- if no byte was read twice, then only 29% of the allocated bytes, which means that at least 71% of the bytes were never read! This suggests that the blocks are being underutilized and might be worth optimizing.

The `Writes` lines is similar to the `Reads` line. In this case, at most 38% of the bytes are ever written, and at least 62% of the bytes were never written.

The `Reads` and `Writes` measurements suggest that the blocks are being under-utilised and might be worth optimizing. Having said that, this kind of under-utilisation is common in data structures that grow, such as vectors and hash tables, and isn't always fixable.

### 10.3.2.4. Leaf Nodes

This is a leaf node:

```

PP 1.1.1.1/2 {
  Total:      31,460,928 bytes (2.32%, 1,565.9/Minstr) in 262,171 blocks (4.41%, 13.05/M
  Max:        16,779,136 bytes in 65,543 blocks, avg size 256 bytes
  At t-gmax:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:   0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:      5,964,704 bytes (0.11%, 296.88/Minstr), 0.19/byte

```

```

Writes:      10,487,200 bytes (0.51%, 521.98/Minstr), 0.33/byte
Allocated at {
  ^1: 0x95CACC9: alloc (alloc.rs:72)
  ^2: 0x95CACC9: alloc (alloc.rs:148)
  ^3: 0x95CACC9: reserve_internal<syntax::tokenstream::TokenStream,alloc::alloc::Global> (raw_
  ^4: 0x95CACC9: reserve<syntax::tokenstream::TokenStream,alloc::alloc::Global> (raw_
  ^5: 0x95CACC9: reserve<syntax::tokenstream::TokenStream> (vec.rs:460)
  ^6: 0x95CACC9: push<syntax::tokenstream::TokenStream> (vec.rs:989)
  ^7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
  ^8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::String
  ^9: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
 ^10: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::String
 #11: 0x95CAC39: parse_token_trees_until_close_delim (tokentrees.rs:26)
 #12: 0x95CAC39: syntax::parse::lexer::tokentrees::<impl syntax::parse::lexer::String
}
}

```

The 1.1.1.1/2 indicates that this node is a great-grandchild of the root; is the first grandchild of the node in the previous example; and has no children.

Leaf nodes contain an additional Max line, indicating the peak memory use for the blocks covered by this PP. (This peak may have occurred at a time other than t-gmax.) In this case, 31,460,298 bytes were allocated from this PP, but the maximum size alive at once was 16,779,136 bytes.

Stack frames that begin with a ^ rather than a # are copied from ancestor nodes. (In this example, the first 8 frames are identical to those from the node in the previous example.) These frames could be found by tracing back through ancestor nodes, but that can be annoying, which is why they are duplicated. This also means that each node makes complete sense on its own.

### 10.3.2.5. Access Counts

If all blocks covered by a PP node have the same size, an additional Accesses field will be present. It indicates how the reads and writes within these blocks were distributed. For example:

```

Total:      8,388,672 bytes (0.62%, 417.53/Minstr) in 262,146 blocks (4.41%, 13.05/Minstr)
At t-gmax: 8,388,672 bytes (1.98%) in 262,146 blocks (16.64%), avg size 32 bytes
At t-end:   0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
Reads:      9,109,682 bytes (0.17%, 453.41/Minstr), 1.09/byte
Writes:     7,340,088 bytes (0.36%, 365.34/Minstr), 0.88/byte
Accesses: {
  [ 0] 65547 7 8 4 65529 # # # 16 # # # 12 # # # # # # # # # # 65542 # # # - - - -
}

```

Every block covered by this PP was 32 bytes. Within all of those blocks, byte 0 was accessed (read or written) 65,547 times, byte 1 was accessed 7 times, byte 2 was accessed 8 times, and so on.

The ditto symbol (#) means "same access count as the previous byte".

A dash (-) means "zero". (It is used instead of 0 because it makes unaccessed regions more easily identifiable.)

The infinity symbol (#, not present in this example) means "exceeded the maximum tracked count".

Block layout can often be inferred from counts. For example, these blocks probably have four separate byte-sized fields, followed by a four-byte field, and so on.

The size of the blocks that measure and display access counts is limited to 1024 bytes. This is done to limit the performance overhead and also to keep the size of the generated output reasonable. However, it is possible to override this limit using client requests. The use-case for this is to first run DHAT normally, and then identify any large blocks that you would like to further investigate with access count histograms. The client request is declared

in `dhat/dhat.h` and is called `DHAT_HISTOGRAM_MEMORY`. The macro should be placed immediately after the call to the allocator, and use the pointer returned by the allocator.

```
// LargeStruct bigger than 1024 bytes
struct LargeStruct* ls = malloc(sizeof(struct LargeStruct));
DHAT_HISTOGRAM_MEMORY(ls);
```

The memory that can be profiled in this way with user requests has a further upper limit of 25kbytes. Be aware that the access counts will all be set to zero. This means that the access counts will not include any reads or writes performed during initialisation. An example where this will happen are uses of C++ `new` with user-defined constructors.

Access counts can be useful for identifying data alignment holes or other layout inefficiencies.

### 10.3.2.6. Aggregate Nodes

The PP tree is very large and many nodes represent tiny numbers of blocks and bytes. Therefore, DHAT's viewer aggregates insignificant nodes like this:

```
PP 1.14.2/2 {
  Total:      5,175 blocks (0.09%, 0.26/Minstr)
  Allocated at {
    [5 insignificant]
  }
}
```

Much of the detail is stripped away, leaving only basic measurements, along with an indication of how many nodes were aggregated together (5 in this case).

### 10.3.3. The Output Footer

Below the PP tree is a line like this:

```
PP significance threshold: total >= 59,434.17 blocks (1%)
```

It shows the function used to determine if a PP node is significant. All nodes that don't satisfy this function are aggregated. It is occasionally useful if you don't understand why a PP node has been aggregated. The exact threshold depends on the sort metric (see below).

Finally, the bottom of the page shows a legend that explains some of the terms, abbreviations and symbols used in the output.

### 10.3.4. Sort Metrics

The order in which sub-trees are sorted can be changed via the "Sort metric" drop-down menu at the top of DHAT's viewer. Different sort metrics can be useful for finding different things. Some sort metrics also incorporate some filtering, so that only nodes meeting a particular criteria are shown.

Total (bytes)

The total number of bytes allocated during the execution. Highly useful for evaluating heap churn, though not quite as useful as "Total (blocks)".

Total (blocks)

The total number of blocks allocated during the execution. Highly useful for evaluating heap churn; reducing the number of calls to the allocator can significantly speed up a program. This is the default sort metric.

**Total (blocks), tiny**

Like "Total (blocks)", but shows only very small blocks. Moderately useful, because such blocks are often easy to avoid allocating.

**Total (blocks), short-lived**

Like "Total (blocks)", but shows only very short-lived blocks. Moderately useful, because such blocks are often easy to avoid allocating.

**Total (bytes), zero reads or zero writes**

Like "Total (bytes)", but shows only blocks that are never read or never written to (or both). Highly useful, because such blocks indicate poor use of memory and are often easy to avoid allocating. For example, sometimes a block is allocated and written to but then only read if a condition C is true; in that case, it may be possible to delay creating the block until condition C is true. Alternatively, sometimes blocks are created and never used; such blocks are trivial to remove.

**Total (blocks), zero reads or zero writes**

Like "Total (bytes), zero reads or zero writes" but for blocks. Highly useful.

**Total (bytes), low-access**

Like "Total (bytes)", but shows only blocks that have low numbers of reads or low numbers of writes (or both). Moderately useful, because such blocks indicate poor use of memory.

**Total (blocks), low-access**

Like "Total (bytes), low-access", but for blocks.

**At t-gmax (bytes)**

This shows the breakdown of memory at the point of peak heap memory usage. Highly useful for reducing peak memory usage.

**At t-end (bytes)**

This shows the breakdown of memory at program termination. Highly useful for identifying process-lifetime leaks.

**Reads (bytes)**

The number of bytes read within heap blocks. Occasionally useful.

**Reads (bytes), high-access**

Like "Reads (bytes)", but only shows blocks with high read ratios. Occasionally useful for identifying hot areas of memory.

**Writes (bytes)**

Like "Reads (bytes)", but for writes. Occasionally useful.

**Writes (bytes), high-access**

Like "Reads (bytes), high-access", but for writes. Occasionally useful.

The values within a node that represent the chosen sort metric are shown in bold, so they stand out.

Here is part of a PP node found with "Total (blocks), tiny", showing blocks with an average size of only 8.67 bytes:

```
Total:      3,407,848 bytes (0.25%, 169.62/Minstr) in 393,214 blocks (6.62%, 19.57/Minstr)
```

Here is part of a PP node found with "Total (blocks), short-lived", showing blocks with an average lifetime of only 181.75 instructions:

```
Total:      23,068,584 bytes (1.7%, 1,148.19/Minstr) in 262,143 blocks (4.41%, 13.05/Minstr)
```

Here is an example of a PP identified with "Total (blocks), zero reads or zero writes", showing blocks that are allocated but never touched:

```
Total:      7,339,920 bytes (0.54%, 365.33/Minstr) in 262,140 blocks (4.41%, 13.05/Minstr)
Max:        3,669,960 bytes in 131,070 blocks, avg size 28 bytes
At t-gmax:  3,336,400 bytes (0.79%) in 119,157 blocks (7.56%), avg size 28 bytes
At t-end:    0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
Reads:       0 bytes (0%, 0/Minstr), 0/byte
Writes:      0 bytes (0%, 0/Minstr), 0/byte
```

All the blocks identified by these PPs are good candidates for optimization.

## 10.4. Treatment of realloc

`realloc` is a tricky function and there are several different ways that DHAT could handle it.

Imagine a `malloc(100)` call followed by a `realloc(200)` call. This combination is considered to add two to the total block count, and 300 bytes to the total bytes count. (An alternative would be to only add one to the total block count, and 200 bytes to the total bytes count, as if a single `malloc(200)` call had occurred. While this would be defensible from a semantic point of view, it is silly from an operational point of view, because making two calls to allocator functions is more expensive than one call, and DHAT is a profiler that aims to help with runtime costs.)

Furthermore, the implicit copying of the 100 bytes is added to the reads and writes counts. Without this, the read and write counts would be under-measured and misleading.

However, DHAT only increases the current heap size by 100 bytes for this combination, and does not change the current block count. (As opposed to increasing the current heap size by 200 bytes and then decreasing it by 100 bytes.) As a result, it can only increase the global heap peak (if indeed, this results in a new peak) by 100 bytes.

Finally, the program point assigned to the block allocated by the `malloc(100)` call is retained once the block is reallocated. Which means that all 300 bytes are attributed to that program point, and no separate program point is created for the `realloc(200)` call. This may be surprising, but it has one large benefit.

Imagine some code that starts with an empty buffer, and then gradually adds data to that buffer from numerous different points in the code, reallocating the buffer each time it gets full. (E.g. code generation in a compiler might work this way.) With the described approach, the first heap block and all subsequent heap blocks are attributed to the same program point. While this is something of a lie -- the first program point isn't actually responsible for the other allocations -- it is arguably better than having the program points spread around in a distribution that unpredictably depends on whenever the reallocations were triggered.

## 10.5. Copy profiling

If DHAT is invoked with `--mode=copy`, instead of profiling heap operations (allocations and deallocations), it profiles copy operations, such as `memcpy`, `memmove`, `strcpy`, and `bcopy`. This is sometimes useful.

Here is an example PP node from this mode:

```
PP 1.1.2/5 (4 children) {
```

```
Total:      1,210,925 bytes (10.03%, 4,358.66/Minstr) in 112,717 blocks (35.2%, 405.72/Minstr)
Copied at {
  ^1: 0x4842524: memmove (vg_replace_strmem.c:1289)
  #2: 0x1F0A0D: copy_nonoverlapping<u8> (intrinsics.rs:1858)
  #3: 0x1F0A0D: copy_from_slice<u8> (mod.rs:2524)
  #4: 0x1F0A0D: spec_extend<u8> (vec.rs:2227)
  #5: 0x1F0A0D: extend_from_slice<u8> (vec.rs:1619)
  #6: 0x1F0A0D: push_str (string.rs:821)
  #7: 0x1F0A0D: write_str (string.rs:2418)
  #8: 0x1F0A0D: <&mut W as core::fmt::Write>::write_str (mod.rs:195)
}
```

It is very similar to the PP nodes for heap profiling, but with less information, because copy profiling doesn't involve any tracking of memory regions with lifetimes.

## 10.6. Ad hoc profiling

If DHAT is invoked with `--mode=ad-hoc`, instead of profiling heap operations (allocations and deallocations), it profiles calls to the `DHAT_AD_HOC_EVENT` client request, which is declared in `dhat/dhat.h`.

Here is an example PP node from this mode:

```
PP 1.1.1.1/2 {
  Total:      30 units (17.65%, 115.97/Minstr) in 1 events (14.29%, 3.87/Minstr), avg si
  Occurred at {
    ^1: 0x109407: g (ad-hoc.c:4)
    ^2: 0x109425: f (ad-hoc.c:8)
    #3: 0x109497: main (ad-hoc.c:14)
  }
}
```

This kind of profiling is useful when you know a code path is hot but you want to know more about it.

For example, you might want to know which callsites of a hot function account for most of the calls. You could put a `DHAT_AD_HOC_EVENT(1);` call at the start of that function.

Alternatively, you might want to know the typical length of a vector in a hot location. You could put a `DHAT_AD_HOC_EVENT(len);` call at the appropriate location, when `len` is the length of the vector.

## 10.7. DHAT Command-line Options

DHAT-specific command-line options are:

`--dhat-out-file=<file>`

Write the profile data to `file` rather than to the default output file, `dhat.out.<pid>`. The `%p` and `%q` format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

`--mode=<heap|copy|ad-hoc> [default: heap]`

The profiling mode: heap profiling, copy profiling, or ad hoc profiling.

Note that stacks by default have 12 frames. This may be more than necessary, in which case the `--num-callers` flag can be used to reduce the number, which may make DHAT run slightly faster.

# 11. Lackey: an example tool

To use this tool, you must specify `--tool=lackey` on the Valgrind command line.

## 11.1. Overview

Lackey is a simple Valgrind tool that does various kinds of basic program measurement. It adds quite a lot of simple instrumentation to the program's code. It is primarily intended to be of use as an example tool, and consequently emphasises clarity of implementation over performance.

## 11.2. Lackey Command-line Options

Lackey-specific command-line options are:

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

When enabled, Lackey prints the following statistics and information about the execution of the client program:

1. The number of calls to the function specified by the `--fnname` option (the default is `main`). If the program has had its symbols stripped, the count will always be zero.
2. The number of conditional branches encountered and the number and proportion of those taken.
3. The number of superblocks entered and completed by the program. Note that due to optimisations done by the JIT, this is not at all an accurate value.
4. The number of guest (x86, amd64, ppc, etc.) instructions and IR statements executed. IR is Valgrind's RISC-like intermediate representation via which all instrumentation is done.
5. Ratios between some of these counts.
6. The exit code of the client program.

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

When enabled, Lackey prints a table containing counts of loads, stores and ALU operations, differentiated by their IR types. The IR types are identified by their IR name ("I1", "I8", ... "I128", "F32", "F64", and "V128").

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

When enabled, Lackey prints the size and address of almost every memory access made by the program. See the comments at the top of the file `lackey/lk_main.c` for details about the output format, how it works, and inaccuracies in the address trace. Note that this option produces immense amounts of output.

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

When enabled, Lackey prints out the address of every superblock (a single entry, multiple exit, linear chunk of code) executed by the program. This is primarily of interest to Valgrind developers. See the comments at the top of the file `lackey/lk_main.c` for details about the output format. Note that this option produces large amounts of output.

`--fnname=<name> [default: main]`

Changes the function for which calls are counted when `--basic-counts=yes` is specified.

## 12. Nulgrind: the minimal Valgrind tool

To use this tool, you must specify `--tool=none` on the Valgrind command line.

### 12.1. Overview

Nulgrind is the simplest possible Valgrind tool. It performs no instrumentation or analysis of a program, just runs it normally. It is mainly of use for Valgrind's developers for debugging and regression testing.

Nonetheless you can run programs with Nulgrind. They will run roughly 5 times more slowly than normal, for no useful effect. Note that you need to use the option `--tool=none` to run Nulgrind (ie. not `--tool=nulgrind`).

# 13. BBV: an experimental basic block vector generation tool

To use this tool, you must specify `--tool=exp-bbv` on the Valgrind command line.

## 13.1. Overview

A basic block is a linear section of code with one entry point and one exit point. A *basic block vector* (BBV) is a list of all basic blocks entered during program execution, and a count of how many times each basic block was run.

BBV is a tool that generates basic block vectors for use with the [SimPoint](#) analysis tool. The SimPoint methodology enables speeding up architectural simulations by only running a small portion of a program and then extrapolating total behavior from this small portion. Most programs exhibit phase-based behavior, which means that at various times during execution a program will encounter intervals of time where the code behaves similarly to a previous interval. If you can detect these intervals and group them together, an approximation of the total program behavior can be obtained by only simulating a bare minimum number of intervals, and then scaling the results.

In computer architecture research, running a benchmark on a cycle-accurate simulator can cause slowdowns on the order of 1000 times, making it take days, weeks, or even longer to run full benchmarks. By utilizing SimPoint this can be reduced significantly, usually by 90-95%, while still retaining reasonable accuracy.

A more complete introduction to how SimPoint works can be found in the paper "Automatically Characterizing Large Scale Program Behavior" by T. Sherwood, E. Perelman, G. Hamerly, and B. Calder.

## 13.2. Using Basic Block Vectors to create SimPoints

To quickly create a basic block vector file, you will call Valgrind like this:

```
valgrind --tool=exp-bbv /bin/ls
```

In this case we are running on `/bin/ls`, but this can be any program. By default a file called `bb.out.PID` will be created, where PID is replaced by the process ID of the running process. This file contains the basic block vector. For long-running programs this file can be quite large, so it might be wise to compress it with gzip or some other compression program.

To create actual SimPoint results, you will need the SimPoint utility, available from the [SimPoint webpage](#). Assuming you have downloaded SimPoint 3.2 and compiled it, create SimPoint results with a command like the following:

```
./SimPoint.3.2/bin/simpoint -inputVectorsGzipped \  
-loadFVFile bb.out.1234.gz \  
-k 5 -saveSimpoints results.simpts \  
-saveSimpointWeights results.weights
```

where `bb.out.1234.gz` is your compressed basic block vector file generated by BBV.

The SimPoint utility does random linear projection using 15-dimensions, then does k-mean clustering to calculate which intervals are of interest. In this example we specify 5 intervals with the `-k 5` option.

The outputs from the SimPoint run are the `results.simpts` and `results.weights` files. The first holds the 5 most relevant intervals of the program. The second holds the weight to scale each interval by when extrapolating full-program behavior. The intervals and the weights can be used in conjunction with a simulator that supports

fast-forwarding; you fast-forward to the interval of interest, collect stats for the desired interval length, then use statistics gathered in conjunction with the weights to calculate your results.

## 13.3. BBV Command-line Options

BBV-specific command-line options are:

`--bb-out-file=<name> [default: bb.out.%p]`

This option selects the name of the basic block vector file. The %p and %q format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

`--pc-out-file=<name> [default: pc.out.%p]`

This option selects the name of the PC file. This file holds program counter addresses and function name info for the various basic blocks. This can be used in conjunction with the basic block vector file to fast-forward via function names instead of just instruction counts. The %p and %q format specifiers can be used to embed the process ID and/or the contents of an environment variable in the name, as is the case for the core option `--log-file`.

`--interval-size=<number> [default: 100000000]`

This option selects the size of the interval to use. The default is 100 million instructions, which is a commonly used value. Other sizes can be used; smaller intervals can help programs with finer-grained phases. However smaller interval size can lead to accuracy issues due to warm-up effects (When fast-forwarding the various architectural features will be un-initialized, and it will take some number of instructions before they "warm up" to the state a full simulation would be at without the fast-forwarding. Large interval sizes tend to mitigate this.)

`--instr-count-only [default: no]`

This option tells the tool to only display instruction count totals, and to not generate the actual basic block vector file. This is useful for debugging, and for gathering instruction count info without generating the large basic block vector files.

## 13.4. Basic Block Vector File Format

The Basic Block Vector is dumped at fixed intervals. This is commonly done every 100 million instructions; the `--interval-size` option can be used to change this.

The output file looks like this:

```
T:45:1024 :189:99343
T:11:78573 :15:1353 :56:1
T:18:45 :12:135353 :56:78 314:4324263
```

Each new interval starts with a T. This is followed on the same line by a series of basic block and frequency pairs, one for each basic block that was entered during the interval. The format for each block/frequency pair is a colon, followed by a number that uniquely identifies the basic block, another colon, and then the frequency (which is the number of times the block was entered, multiplied by the number of instructions in the block). The pairs are separated from each other by a space.

The frequency count is multiplied by the number of instructions that are in the basic block, in order to weigh the count so that instructions in small basic blocks aren't counted as more important than instructions in large basic blocks.

The SimPoint program only processes lines that start with a "T". All other lines are ignored. Traditionally comments are indicated by starting a line with a "#" character. Some other BBV generation tools, such as PinPoints,

generate lines beginning with letters other than "T" to indicate more information about the program being run. We do not generate these, as the SimPoint utility ignores them.

## 13.5. Implementation

Valgrind provides all of the information necessary to create BBV files. In the current implementation, all instructions are instrumented. This is slower (by approximately a factor of two) than a method that instruments at the basic block level, but there are some complications (especially with rep prefix detection) that make that method more difficult.

Valgrind actually provides instrumentation at a superblock level. A superblock has one entry point but unlike basic blocks can have multiple exit points. Once a branch occurs into the middle of a block, it is split into a new basic block. Because Valgrind cannot produce "true" basic blocks, the generated BBV vectors will be different than those generated by other tools. In practice this does not seem to affect the accuracy of the SimPoint results. We do internally force the `--vex-guest-chase=no` option to Valgrind which forces a more basic-block-like behavior.

When a superblock is run for the first time, it is instrumented with our BBV routine. A block info (bbInfo) structure is allocated which holds the various information and statistics for the block. A unique block ID is assigned to the block, and then the structure is placed into an ordered set. Then each native instruction in the block is instrumented to call an instruction counting routine with a pointer to the block info structure as an argument.

At run-time, our instruction counting routines are called once per native instruction. The relevant block info structure is accessed and the block count and total instruction count is updated. If the total instruction count overflows the interval size then we walk the ordered set, writing out the statistics for any block that was accessed in the interval, then resetting the block counters to zero.

On the x86 and amd64 architectures the counting code has extra code to handle rep-prefixed string instructions. This is because actual hardware counts a rep-prefixed instruction as one instruction, while a naive Valgrind implementation would count it as many (possibly hundreds, thousands or even millions) of instructions. We handle rep-prefixed instructions specially, in order to make the results match those obtained with hardware performance counters.

BBV also counts the `fildcw` instruction. This instruction is used on x86 machines in various ways; it is most commonly found when converting floating point values into integers. On Pentium 4 systems the retired instruction performance counter counts this instruction as two instructions (all other known processors only count it as one). This can affect results when using SimPoint on Pentium 4 systems. We provide the `fildcw` count so that users can evaluate whether it will impact their results enough to avoid using Pentium 4 machines for their experiments. It would be possible to add an option to this tool that mimics the double-counting so that the generated BBV files would be usable for experiments using hardware performance counters on Pentium 4 systems.

## 13.6. Threaded Executable Support

BBV supports threaded programs. When a program has multiple threads, an additional basic block vector file is created for each thread (each additional file is the specified filename with the thread number appended at the end).

There is no official method of using SimPoint with threaded workloads. The most common method is to run SimPoint on each thread's results independently, and use some method of deterministic execution to try to match the original workload. This should be possible with the current BBV.

## 13.7. Validation

BBV has been tested on x86, amd64, and ppc32 platforms. An earlier version of BBV was tested in detail using hardware performance counters, this work is described in a paper from the HiPEAC'08 conference, "Using Dynamic Binary Instrumentation to Generate Multi-Platform SimPoints: Methodology and Accuracy" by V.M. Weaver and S.A. McKee.

## 13.8. Performance

Using this program slows down execution by roughly a factor of 40 over native execution. This varies depending on the machine used and the benchmark being run. On the SPEC CPU 2000 benchmarks running on a 3.4GHz Pentium D processor, the slowdown ranges from 24x (mcf) to 340x (vortex.2).

# Valgrind FAQ

**Release 3.23.0.GIT ?? Apr 2024**

**Copyright © 2000-2022 [Valgrind Developers](#)**

Email: [valgrind@valgrind.org](mailto:valgrind@valgrind.org)

## Table of Contents

Valgrind Frequently Asked Questions .....	1
---	---

# Valgrind Frequently Asked Questions

1. Background .....	1
1.1. How do you pronounce "Valgrind"? .....	1
1.2. Where does the name "Valgrind" come from? .....	1
2. Compiling, installing and configuring .....	2
2.1. When building Valgrind, 'make' dies partway with an assertion failure, something like this: .....	2
2.2. When building Valgrind, 'make' fails with this: .....	2
3. Valgrind aborts unexpectedly .....	2
3.1. Programs run OK on Valgrind, but at exit produce a bunch of errors involving __libc_freeres and then die with a segmentation fault. ....	2
3.2. My (buggy) program dies like this: .....	2
3.3. My program dies, printing a message like this along the way: .....	2
3.4. I tried running a Java program (or another program that uses a just-in-time compiler) under Valgrind but something went wrong. Does Valgrind handle such programs? .....	3
4. Valgrind behaves unexpectedly .....	3
4.1. My program uses the C++ STL and string classes. Valgrind reports 'still reachable' memory leaks involving these classes at the exit of the program, but there should be none. ....	3
4.2. The stack traces given by Memcheck (or another tool) aren't helpful. How can I improve them? .....	3
4.3. The stack traces given by Memcheck (or another tool) seem to have the wrong function name in them. What's happening? .....	4
4.4. My program crashes normally, but doesn't under Valgrind, or vice versa. What's happening? .....	5
4.5. Memcheck doesn't report any errors and I know my program has errors. ....	5
4.6. Why doesn't Memcheck find the array overruns in this program? .....	5
5. Miscellaneous .....	6
5.1. I tried writing a suppression but it didn't work. Can you write my suppression for me? .....	6
5.2. With Memcheck's memory leak detector, what's the difference between "definitely lost", "indirectly lost", "possibly lost", "still reachable", and "suppressed"? .....	6
5.3. Memcheck's uninitialised value errors are hard to track down, because they are often reported some time after they are caused. Could Memcheck record a trail of operations to better link the cause to the effect? Or maybe just eagerly report any copies of uninitialised memory values? .....	6
5.4. Is it possible to attach Valgrind to a program that is already running? .....	7
6. How To Get Further Assistance .....	7
6.1. Where can I get more help? .....	7

## 1. Background

### 1.1. How do you pronounce "Valgrind"?

The "Val" as in the word "value". The "grind" is pronounced with a short 'i' -- ie. "grinned" (rhymes with "tinned") rather than "grined" (rhymes with "find").

Don't feel bad: almost everyone gets it wrong at first.

### 1.2. Where does the name "Valgrind" come from?

From Nordic mythology. Originally (before release) the project was named Heimdall, after the watchman of the Nordic gods. He could "see a hundred miles by day or night, hear the grass growing, see the wool growing on a sheep's back", etc. This would have been a great name, but it was already taken by a security package "Heimdall".

Keeping with the Nordic theme, Valgrind was chosen. Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard). Over this entrance there resides a wolf and over it there is the head of a boar and on it perches a huge eagle, whose eyes can see to the far regions of the nine worlds. Only those judged worthy by the guardians are allowed to pass through Valgrind. All others are refused entrance.

It's not short for "value grinder", although that's not a bad guess.

## 2. Compiling, installing and configuring

- 2.1. When building Valgrind, 'make' dies partway with an assertion failure, something like this:

```
% make: expand.c:489: allocated_variable_append:
Assertion 'current_variable_set_list->next != 0' failed.
```

It's probably a bug in 'make'. Some, but not all, instances of version 3.79.1 have this bug, see [this](#). Try upgrading to a more recent version of 'make'. Alternatively, we have heard that unsetting the CFLAGS environment variable avoids the problem.

- 2.2. When building Valgrind, 'make' fails with this:

```
/usr/bin/ld: cannot find -lc
collect2: ld returned 1 exit status
```

You need to install the glibc-static-devel package.

## 3. Valgrind aborts unexpectedly

- 3.1. Programs run OK on Valgrind, but at exit produce a bunch of errors involving `__libc_freeres` and then die with a segmentation fault.

When the program exits, Valgrind runs the procedure `__libc_freeres` in glibc. This is a hook for memory debuggers, so they can ask glibc to free up any memory it has used. Doing that is needed to ensure that Valgrind doesn't incorrectly report space leaks in glibc.

The problem is that running `__libc_freeres` in older glibc versions causes this crash.

Workaround for 1.1.X and later versions of Valgrind: use the `--run-libc-freeres=no` option. You may then get space leak reports for glibc allocations (please don't report these to the glibc people, since they are not real leaks), but at least the program runs.

- 3.2. My (buggy) program dies like this:

```
valgrind: m_mallocfree.c:248 (get_bszB_as_is): Assertion 'bszB_lo == bszB_hi' fail
```

or like this:

```
valgrind: m_mallocfree.c:442 (mk_inuse_bszB): Assertion 'bszB != 0' failed.
```

or otherwise aborts or crashes in `m_mallocfree.c`.

If Memcheck (the memory checker) shows any invalid reads, invalid writes or invalid frees in your program, the above may happen. Reason is that your program may trash Valgrind's low-level memory manager, which then dies with the above assertion, or something similar. The cure is to fix your program so that it doesn't do any illegal memory accesses. The above failure will hopefully go away after that.

- 3.3. My program dies, printing a message like this along the way:

```
vex x86->IR: unhandled instruction bytes: 0x66 0xF 0x2E 0x5
```

One possibility is that your program has a bug and erroneously jumps to a non-code address, in which case you'll get a SIGILL signal. Memcheck may issue a warning just before this happens, but it might not if the jump happens to land in addressable memory.

Another possibility is that Valgrind does not handle the instruction. If you are using an older Valgrind, a newer version might handle the instruction. However, all instruction sets have some obscure, rarely used instructions. Also, on amd64 there are an almost limitless number of combinations of redundant instruction prefixes, many of them undocumented but accepted by CPUs. So Valgrind will still have decoding failures from time to time. If this happens, please file a bug report.

- 3.4.** I tried running a Java program (or another program that uses a just-in-time compiler) under Valgrind but something went wrong. Does Valgrind handle such programs?

Valgrind can handle dynamically generated code, so long as none of the generated code is later overwritten by other generated code. If this happens, though, things will go wrong as Valgrind will continue running its translations of the old code (this is true on x86 and amd64, on PowerPC there are explicit cache flush instructions which Valgrind detects and honours). You should try running with `--smc-check=all` in this case. Valgrind will run much more slowly, but should detect the use of the out-of-date code.

Alternatively, if you have the source code to the JIT compiler you can insert calls to the `VALGRIND_DISCARD_TRANSLATIONS` client request to mark out-of-date code, saving you from using `--smc-check=all`.

Apart from this, in theory Valgrind can run any Java program just fine, even those that use JNI and are partially implemented in other languages like C and C++. In practice, Java implementations tend to do nasty things that most programs do not, and Valgrind sometimes falls over these corner cases.

If your Java programs do not run under Valgrind, even with `--smc-check=all`, please file a bug report and hopefully we'll be able to fix the problem.

## 4. Valgrind behaves unexpectedly

- 4.1.** My program uses the C++ STL and string classes. Valgrind reports 'still reachable' memory leaks involving these classes at the exit of the program, but there should be none.

First of all: relax, it's probably not a bug, but a feature. Many implementations of the C++ standard libraries use their own memory pool allocators. Memory for quite a number of destructed objects is not immediately freed and given back to the OS, but kept in the pool(s) for later re-use. The fact that the pools are not freed at the exit of the program cause Valgrind to report this memory as still reachable. The behaviour not to free pools at the exit could be called a bug of the library though.

Using GCC, you can force the STL to use malloc and to free memory as soon as possible by globally disabling memory caching. Beware! Doing so will probably slow down your program, sometimes drastically.

- With GCC 2.91, 2.95, 3.0 and 3.1, compile all source using the STL with `-D__USE_MALLOC`. Beware! This was removed from GCC starting with version 3.3.
- With GCC 3.2.2 and later, you should export the environment variable `GLIBCPP_FORCE_NEW` before running your program.
- With GCC 3.4 and later, that variable has changed name to `GLIBCXX_FORCE_NEW`.

There are other ways to disable memory pooling: using the `malloc_alloc` template with your objects (not portable, but should work for GCC) or even writing your own memory allocators. But all this goes beyond the scope of this FAQ. Start by reading [http://gcc.gnu.org/onlinedocs/libstdc++/faq/index.html#4\\_4\\_leak](http://gcc.gnu.org/onlinedocs/libstdc++/faq/index.html#4_4_leak) if you absolutely want to do that. But beware: allocators belong to the more messy parts of the STL and people went to great lengths to make the STL portable across platforms. Chances are good that your solution will work on your platform, but not on others.

- 4.2.** The stack traces given by Memcheck (or another tool) aren't helpful. How can I improve them?

If they're not long enough, use `--num-callers` to make them longer.

If they're not detailed enough, make sure you are compiling with `-g` to add debug information. And don't strip symbol tables (programs should be unstripped unless you run 'strip' on them; some libraries ship stripped).

Also, for leak reports involving shared objects, if the shared object is unloaded before the program terminates, Valgrind will discard the debug information and the error message will be full of ??? entries. If you use the option `--keep-debuginfo=yes`, then Valgrind will keep the debug information in order to show the stack traces, at the price of increased memory. An alternate workaround is to avoid calling `dlclose` on these shared objects.

Also, `-fomit-frame-pointer` and `-fstack-check` can make stack traces worse.

Some example sub-traces:

- With debug information and unstripped (best):

```
Invalid write of size 1
  at 0x80483BF: really (malloc1.c:20)
 by 0x8048370: main (malloc1.c:9)
```

- With no debug information, unstripped:

```
Invalid write of size 1
  at 0x80483BF: really (in /auto/homes/njn25/grind/head5/a.out)
 by 0x8048370: main (in /auto/homes/njn25/grind/head5/a.out)
```

- With no debug information, stripped:

```
Invalid write of size 1
  at 0x80483BF: (within /auto/homes/njn25/grind/head5/a.out)
 by 0x8048370: (within /auto/homes/njn25/grind/head5/a.out)
 by 0x42015703: __libc_start_main (in /lib/tls/libc-2.3.2.so)
 by 0x80482CC: (within /auto/homes/njn25/grind/head5/a.out)
```

- With debug information and `-fomit-frame-pointer`:

```
Invalid write of size 1
  at 0x80483C4: really (malloc1.c:20)
 by 0x42015703: __libc_start_main (in /lib/tls/libc-2.3.2.so)
 by 0x80482CC: ??? (start.S:81)
```

- A leak error message involving an unloaded shared object:

```
84 bytes in 1 blocks are possibly lost in loss record 488 of 713
  at 0x1B9036DA: operator new(unsigned) (vg_replace_malloc.c:132)
 by 0x1DB63EEB: ???
 by 0x1DB4B800: ???
 by 0x1D65E007: ???
 by 0x8049EE6: main (main.cpp:24)
```

- 4.3.** The stack traces given by Memcheck (or another tool) seem to have the wrong function name in them. What's happening?

Occasionally Valgrind stack traces get the wrong function names. This is caused by glibc using aliases to effectively give one function two names. Most of the time Valgrind chooses a suitable name, but very

occasionally it gets it wrong. Examples we know of are printing `bcmp` instead of `memcmp`, `index` instead of `strchr`, and `rindex` instead of `strrchr`.

**4.4.** My program crashes normally, but doesn't under Valgrind, or vice versa. What's happening?

When a program runs under Valgrind, its environment is slightly different to when it runs natively. For example, the memory layout is different, and the way that threads are scheduled is different.

Most of the time this doesn't make any difference, but it can, particularly if your program is buggy. For example, if your program crashes because it erroneously accesses memory that is unaddressable, it's possible that this memory will not be unaddressable when run under Valgrind. Alternatively, if your program has data races, these may not manifest under Valgrind.

There isn't anything you can do to change this, it's just the nature of the way Valgrind works that it cannot exactly replicate a native execution environment. In the case where your program crashes due to a memory error when run natively but not when run under Valgrind, in most cases Memcheck should identify the bad memory operation.

**4.5.** Memcheck doesn't report any errors and I know my program has errors.

There are two possible causes of this.

First, by default, Valgrind only traces the top-level process. So if your program spawns children, they won't be traced by Valgrind by default. Also, if your program is started by a shell script, Perl script, or something similar, Valgrind will trace the shell, or the Perl interpreter, or equivalent.

To trace child processes, use the `--trace-children=yes` option.

If you are tracing large trees of processes, it can be less disruptive to have the output sent over the network. Give Valgrind the option `--log-socket=127.0.0.1:12345` (if you want logging output sent to port 12345 on localhost). You can use the `valgrind-listener` program to listen on that port:

```
valgrind-listener 12345
```

Obviously you have to start the listener process first. See the manual for more details.

Second, if your program is statically linked, most Valgrind tools will only work well if they are able to replace certain functions, such as `malloc`, with their own versions. By default, statically linked `malloc` functions are not replaced. A key indicator of this is if Memcheck says:

```
All heap blocks were freed -- no leaks are possible
```

when you know your program calls `malloc`. The workaround is to use the option `--soname-synonyms=somalloc=NONE` or to avoid statically linking your program.

There will also be no replacement if you use an alternative `malloc` library such as `tcmalloc`, `jemalloc`, ... In such a case, the option `--soname-synonyms=somalloc=zzzz` (where `zzzz` is the soname of the alternative `malloc` library) will allow Valgrind to replace the functions.

**4.6.** Why doesn't Memcheck find the array overruns in this program?

```
int static[5];

int main(void)
{
    int stack[5];

    static[5] = 0;
```

```

    stack [5] = 0;

    return 0;
}

```

Unfortunately, Memcheck doesn't do bounds checking on global or stack arrays. We'd like to, but it's just not possible to do in a reasonable way that fits with how Memcheck works. Sorry.

## 5. Miscellaneous

### 5.1. I tried writing a suppression but it didn't work. Can you write my suppression for me?

Yes! Use the `--gen-suppressions=yes` feature to spit out suppressions automatically for you. You can then edit them if you like, eg. combining similar automatically generated suppressions using wildcards like `'*'`.

If you really want to write suppressions by hand, read the manual carefully. Note particularly that C++ function names must be mangled (that is, not demangled).

### 5.2. With Memcheck's memory leak detector, what's the difference between "definitely lost", "indirectly lost", "possibly lost", "still reachable", and "suppressed"?

The details are in the Memcheck section of the user manual.

In short:

- "definitely lost" means your program is leaking memory -- fix those leaks!
- "indirectly lost" means your program is leaking memory in a pointer-based structure. (E.g. if the root node of a binary tree is "definitely lost", all the children will be "indirectly lost".) If you fix the "definitely lost" leaks, the "indirectly lost" leaks should go away.
- "possibly lost" means your program is leaking memory, unless you're doing unusual things with pointers that could cause them to point into the middle of an allocated block; see the user manual for some possible causes. Use `--show-possibly-lost=no` if you don't want to see these reports.
- "still reachable" means your program is probably ok -- it didn't free some memory it could have. This is quite common and often reasonable. Don't use `--show-reachable=yes` if you don't want to see these reports.
- "suppressed" means that a leak error has been suppressed. There are some suppressions in the default suppression files. You can ignore suppressed errors.

### 5.3. Memcheck's uninitialised value errors are hard to track down, because they are often reported some time after they are caused. Could Memcheck record a trail of operations to better link the cause to the effect? Or maybe just eagerly report any copies of uninitialised memory values?

Prior to version 3.4.0, the answer was "we don't know how to do it without huge performance penalties". As of 3.4.0, try using the `--track-origins=yes` option. It will run slower than usual, but will give you extra information about the origin of uninitialised values.

Or if you want to do it the old fashioned way, you can use the client request `VALGRIND_CHECK_VALUE_IS_DEFINED` to help track these errors down -- work backwards from the point where the uninitialised error occurs, checking suspect values until you find the cause. This requires editing, compiling and re-running your program multiple times, which is a pain, but still easier than debugging the problem without Memcheck's help.

As for eager reporting of copies of uninitialised memory values, this has been suggested multiple times. Unfortunately, almost all programs legitimately copy uninitialised memory values around (because compilers pad structs to preserve alignment) and eager checking leads to hundreds of false positives. Therefore Memcheck does not support eager checking at this time.

**5.4.** Is it possible to attach Valgrind to a program that is already running?

No. The environment that Valgrind provides for running programs is significantly different to that for normal programs, e.g. due to different layout of memory. Therefore Valgrind has to have full control from the very start.

It is possible to achieve something like this by running your program without any instrumentation (which involves a slow-down of about 5x, less than that of most tools), and then adding instrumentation once you get to a point of interest. Support for this must be provided by the tool, however, and Callgrind is the only tool that currently has such support. See the instructions on the `callgrind_control` program for details.

## 6. How To Get Further Assistance

**6.1.** Where can I get more help?

Read the appropriate section(s) of the [Valgrind Documentation](#).

[Search](#) the [valgrind-users](#) mailing list archives, using the group name `gmane.comp.debugging.valgrind`.

If you think an answer in this FAQ is incomplete or inaccurate, please e-mail [valgrind@valgrind.org](mailto:valgrind@valgrind.org).

If you have tried all of these things and are still stuck, you can try mailing the [valgrind-users mailing list](#). Note that an email has a better change of being answered usefully if it is clearly written. Also remember that, despite the fact that most of the community are very helpful and responsive to emailed questions, you are probably requesting help from unpaid volunteers, so you have no guarantee of receiving an answer.

# Valgrind Technical Documentation

Release 3.23.0.GIT ?? Apr 2024

Copyright © 2000-2022 [Valgrind Developers](#)

Email: [valgrind@valgrind.org](mailto:valgrind@valgrind.org)

# Table of Contents

1. The Design and Implementation of Valgrind .....	1
2. Writing a New Valgrind Tool .....	2
2.1. Introduction .....	2
2.2. Basics .....	2
2.2.1. How tools work .....	2
2.2.2. Getting the code .....	2
2.2.3. Getting started .....	2
2.2.4. Writing the code .....	3
2.2.5. Initialisation .....	3
2.2.6. Instrumentation .....	4
2.2.7. Finalisation .....	4
2.2.8. Other Important Information .....	4
2.3. Advanced Topics .....	5
2.3.1. Debugging Tips .....	5
2.3.2. Suppressions .....	5
2.3.3. Documentation .....	5
2.3.4. Regression Tests .....	6
2.3.5. Profiling .....	6
2.3.6. Other Makefile Hackery .....	7
2.3.7. The Core/tool Interface .....	7
2.4. Final Words .....	7
3. Callgrind Format Specification .....	8
3.1. Overview .....	8
3.1.1. Basic Structure .....	8
3.1.2. Simple Example .....	8
3.1.3. Associations .....	9
3.1.4. Extended Example .....	9
3.1.5. Name Compression .....	10
3.1.6. Subposition Compression .....	11
3.1.7. Miscellaneous .....	11
3.2. Reference .....	12
3.2.1. Grammar .....	12
3.2.2. Description of Header Lines .....	13
3.2.3. Description of Body Lines .....	15

# 1. The Design and Implementation of Valgrind

A number of academic publications nicely describe many aspects of Valgrind's design and implementation. Online copies of all of them, and others, are available on the [Valgrind publications page](#).

The following paper gives a good overview of Valgrind, and explains how it differs from other dynamic binary instrumentation frameworks such as Pin and DynamoRIO.

- **Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.** Nicholas Nethercote and Julian Seward. **Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.**

The following two papers together give a comprehensive description of how most of Memcheck works. The first paper describes in detail how Memcheck's undefined value error detection (a.k.a. V bits) works. The second paper describes in detail how Memcheck's shadow memory is implemented, and compares it to other alternative approaches.

- **Using Valgrind to detect undefined value errors with bit-precision.** Julian Seward and Nicholas Nethercote. **Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, April 2005.**

**How to Shadow Every Byte of Memory Used by a Program.** Nicholas Nethercote and Julian Seward. **Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007), San Diego, California, USA, June 2007.**

The following paper describes Callgrind.

- **A Tool Suite for Simulation Based Analysis of Memory Access Behavior.** Josef Weidendorfer, Markus Kowarschik and Carsten Trinitis. **Proceedings of the 4th International Conference on Computational Science (ICCS 2004), Krakow, Poland, June 2004.**

The following dissertation describes Valgrind in some detail (many of these details are now out-of-date) as well as Cachegrind, Annelid and Redux. It also covers some underlying theory about dynamic binary analysis in general and what all these tools have in common.

- **Dynamic Binary Analysis and Instrumentation.** Nicholas Nethercote. PhD Dissertation, University of Cambridge, November 2004.

# 2. Writing a New Valgrind Tool

So you want to write a Valgrind tool? Here are some instructions that may help.

## 2.1. Introduction

The key idea behind Valgrind's architecture is the division between its *core* and *tools*.

The core provides the common low-level infrastructure to support program instrumentation, including the JIT compiler, low-level memory manager, signal handling and a thread scheduler. It also provides certain services that are useful to some but not all tools, such as support for error recording, and support for replacing heap allocation functions such as `malloc`.

But the core leaves certain operations undefined, which must be filled by tools. Most notably, tools define how program code should be instrumented. They can also call certain functions to indicate to the core that they would like to use certain services, or be notified when certain interesting events occur. But the core takes care of all the hard work.

## 2.2. Basics

### 2.2.1. How tools work

Tools must define various functions for instrumenting programs that are called by Valgrind's core. They are then linked against Valgrind's core to define a complete Valgrind tool which will be used when the `--tool` option is used to select it.

### 2.2.2. Getting the code

To write your own tool, you'll need the Valgrind source code. You'll need a clone from the git repository for the automake/autoconf build instructions to work. See the information about how to do clone from the repository at [the Valgrind website](#).

### 2.2.3. Getting started

Valgrind uses GNU automake and autoconf for the creation of Makefiles and configuration. But don't worry, these instructions should be enough to get you started even if you know nothing about those tools.

In what follows, all filenames are relative to Valgrind's top-level directory `valgrind/`.

1. Choose a name for the tool, and a two-letter abbreviation that can be used as a short prefix. We'll use `foobar` and `fb` as an example.
2. Make three new directories `foobar/`, `foobar/docs/` and `foobar/tests/`.
3. Create an empty file `foobar/tests/Makefile.am`.
4. Copy `none/Makefile.am` into `foobar/`. Edit it by replacing all occurrences of the strings `"none"`, `"nl_"` and `"nl-"` with `"foobar"`, `"fb_"` and `"fb-"` respectively.
5. Copy `none/nl_main.c` into `foobar/`, renaming it as `fb_main.c`. Edit it by changing the `details` lines in `nl_pre_clo_init` to something appropriate for the tool. These fields are used in the startup message, except for `bug_reports_to` which is used if a tool assertion fails. Also, replace the string `"nl_"` throughout with `"fb_"` again.
6. Edit `Makefile.am`, adding the new directory `foobar` to the `TOOLS` or `EXP_TOOLS` variables.

7. Edit `configure.ac`, adding `foobar/Makefile` and `foobar/tests/Makefile` to the `AC_OUTPUT` list.

8. Run:

```
autogen.sh
./configure --prefix=`pwd`/inst
make
make install
```

It should automake, configure and compile without errors, putting copies of the tool in `foobar/` and `inst/lib/valgrind/`.

9. You can test it with a command like:

```
inst/bin/valgrind --tool=foobar date
```

(almost any program should work; `date` is just an example). The output should be something like this:

```
==738== foobar-0.0.1, a foobarring tool.
==738== Copyright (C) 2002-2017, and GNU GPL'd, by J. Programmer.
==738== Using Valgrind-3.14.0.GIT and LibVEX; rerun with -h for copyright info
==738== Command: date
==738==
Tue Nov 27 12:40:49 EST 2017
==738==
```

The tool does nothing except run the program uninstrumented.

These steps don't have to be followed exactly -- you can choose different names for your source files, and use a different `--prefix` for `./configure`.

Now that we've setup, built and tested the simplest possible tool, onto the interesting stuff...

## 2.2.4. Writing the code

A tool must define at least these four functions:

```
pre_clo_init()
post_clo_init()
instrument()
fini()
```

The names can be different to the above, but these are the usual names. The first one is registered using the macro `VG_DETERMINE_INTERFACE_VERSION`. The last three are registered using the `VG_(basic_tool_funcs)` function.

In addition, if a tool wants to use some of the optional services provided by the core, it may have to define other functions and tell the core about them.

## 2.2.5. Initialisation

Most of the initialisation should be done in `pre_clo_init`. Only use `post_clo_init` if a tool provides command line options and must do some initialisation after option processing takes place ("`clo`" stands for "command line options").

First of all, various "details" need to be set for a tool, using the functions `VG_(details_*)`. Some are all compulsory, some aren't. Some are used when constructing the startup message, `detail_bug_reports_to` is used if `VG_(tool_panic)` is ever called, or a tool assertion fails. Others have other uses.

Second, various "needs" can be set for a tool, using the functions `VG_(needs_*)`. They are mostly booleans, and can be left untouched (they default to `False`). They determine whether a tool can do various things such as: record, report and suppress errors; process command line options; wrap system calls; record extra information about heap blocks; etc.

For example, if a tool wants the core's help in recording and reporting errors, it must call `VG_(needs_tool_errors)` and provide definitions of eight functions for comparing errors, printing out errors, reading suppressions from a suppressions file, etc. While writing these functions requires some work, it's much less than doing error handling from scratch because the core is doing most of the work.

Third, the tool can indicate which events in core it wants to be notified about, using the functions `VG_(track_*)`. These include things such as heap blocks being allocated, the stack pointer changing, a mutex being locked, etc. If a tool wants to know about this, it should provide a pointer to a function, which will be called when that event happens.

For example, if the tool want to be notified when a new heap block is allocated, it should call `VG_(track_new_mem_heap)` with an appropriate function pointer, and the assigned function will be called each time this happens.

More information about "details", "needs" and "trackable events" can be found in `include/pub_tool_tooliface.h`.

## 2.2.6. Instrumentation

`instrument` is the interesting one. It allows you to instrument *VEX IR*, which is Valgrind's RISC-like intermediate language. VEX IR is described in the comments of the header file `VEX/pub/libvex_ir.h`.

The easiest way to instrument VEX IR is to insert calls to C functions when interesting things happen. See the tool "Lackey" (`lackey/lk_main.c`) for a simple example of this, or Cachegrind (`cachegrind/cg_main.c`) for a more complex example.

## 2.2.7. Finalisation

This is where you can present the final results, such as a summary of the information collected. Any log files should be written out at this point.

## 2.2.8. Other Important Information

Please note that the core/tool split infrastructure is quite complex and not brilliantly documented. Here are some important points, but there are undoubtedly many others that I should note but haven't thought of.

The files `include/pub_tool_*.h` contain all the types, macros, functions, etc. that a tool should (hopefully) need, and are the only `.h` files a tool should need to `#include`. They have a reasonable amount of documentation in it that should hopefully be enough to get you going.

Note that you can't use anything from the C library (there are deep reasons for this, trust us). Valgrind provides an implementation of a reasonable subset of the C library, details of which are in `pub_tool_libc*.h`.

When writing a tool, in theory you shouldn't need to look at any of the code in Valgrind's core, but in practice it might be useful sometimes to help understand something.

The `include/pub_tool_basics.h` and `VEX/pub/libvex_basictypes.h` files have some basic types that are widely used.

Ultimately, the tools distributed (Memcheck, Cachegrind, Lackey, etc.) are probably the best documentation of all, for the moment.

The `VG_` macro is used heavily. This just prepends a longer string in front of names to avoid potential namespace clashes. It is defined in `include/pub_tool_basics.h`.

There are some assorted notes about various aspects of the implementation in `docs/internals/`. Much of it isn't that relevant to tool-writers, however.

## 2.3. Advanced Topics

Once a tool becomes more complicated, there are some extra things you may want/need to do.

### 2.3.1. Debugging Tips

Writing and debugging tools is not trivial. Here are some suggestions for solving common problems.

If you are getting segmentation faults in C functions used by your tool, the usual GDB command:

```
gdb <prog> core
```

usually gives the location of the segmentation fault.

If you want to debug C functions used by your tool, there are instructions on how to do so in the file `README_DEVELOPERS`.

If you are having problems with your VEX IR instrumentation, it's likely that GDB won't be able to help at all. In this case, Valgrind's `--trace-flags` option is invaluable for observing the results of instrumentation.

If you just want to know whether a program point has been reached, using the `OINK` macro (in `include/pub_tool_libcprint.h`) can be easier than using GDB.

The other debugging command line options can be useful too (run `valgrind --help-debug` for the list).

### 2.3.2. Suppressions

If your tool reports errors and you want to suppress some common ones, you can add suppressions to the suppression files. The relevant files are `*.supp`; the final suppression file is aggregated from these files by combining the relevant `.supp` files depending on the versions of linux, X and glibc on a system.

Suppression types have the form `tool_name:suppression_name`. The `tool_name` here is the name you specify for the tool during initialisation with `VG_(details_name)`.

### 2.3.3. Documentation

If you are feeling conscientious and want to write some documentation for your tool, please use XML as the rest of Valgrind does. The file `docs/README` has more details on getting the XML toolchain to work; this can be difficult, unfortunately.

To write the documentation, follow these steps (using `foobar` as the example tool name again):

1. The docs go in `foobar/docs/`, which you will have created when you started writing the tool.
2. Copy the XML documentation file for the tool Nulgrind from `none/docs/nl-manual.xml` to `foobar/docs/`, and rename it to `foobar/docs/fb-manual.xml`.

**Note:** there is a tetex bug involving underscores in filenames, so don't use `'_'`.

3. Write the documentation. There are some helpful bits and pieces on using XML markup in `docs/xml/xml_help.txt`.
4. Include it in the User Manual by adding the relevant entry to `docs/xml/manual.xml`. Copy and edit an existing entry.
5. Include it in the man page by adding the relevant entry to `docs/xml/valgrind-manpage.xml`. Copy and edit an existing entry.
6. Validate `foobar/docs/fb-manual.xml` using the following command from within `docs/`:

```
make valid
```

You may get errors that look like this:

```
./xml/index.xml:5: element chapter: validity error : No declaration for
attribute base of element chapter
```

Ignore (only) these -- they're not important.

Because the XML toolchain is fragile, it is important to ensure that `fb-manual.xml` won't break the documentation set build. Note that just because an XML file happily transforms to html does not necessarily mean the same holds true for pdf/ps.

7. You can (re-)generate the HTML docs while you are writing `fb-manual.xml` to help you see how it's looking. The generated files end up in `docs/html/`. Use the following command, within `docs/`:

```
make html-docs
```

8. When you have finished, try to generate PDF and PostScript output to check all is well, from within `docs/`:

```
make print-docs
```

Check the output `.pdf` and `.ps` files in `docs/print/`.

Note that the toolchain is even more fragile for the print docs, so don't feel too bad if you can't get it working.

## 2.3.4. Regression Tests

Valgrind has some support for regression tests. If you want to write regression tests for your tool:

1. The tests go in `foobar/tests/`, which you will have created when you started writing the tool.
2. Write `foobar/tests/Makefile.am`. Use `memcheck/tests/Makefile.am` as an example.
3. Write the tests, `.vgtest` test description files, `.stdout.exp` and `.stderr.exp` expected output files. (Note that Valgrind's output goes to `stderr`.) Some details on writing and running tests are given in the comments at the top of the testing script `tests/vg_regtest`.
4. Write a filter for `stderr` results `foobar/tests/filter_stderr`. It can call the existing filters in `tests/`. See `memcheck/tests/filter_stderr` for an example; in particular note the `$dir` trick that ensures the filter works correctly from any directory.

## 2.3.5. Profiling

Lots of profiling tools have trouble running Valgrind. For example, trying to use `gprof` is hopeless.

Probably the best way to profile a tool is with OProfile on Linux.

You can also use Cachegrind on it. Read `README_DEVELOPERS` for details on running Valgrind under Valgrind; it's a bit fragile but can usually be made to work.

## 2.3.6. Other Makefile Hackery

If you add any directories under `foobar/`, you will need to add an appropriate `Makefile.am` to it, and add a corresponding entry to the `AC_OUTPUT` list in `configure.ac`.

If you add any scripts to your tool (see Cachegrind for an example) you need to add them to the `bin_SCRIPTS` variable in `foobar/Makefile.am` and possible also to the `AC_OUTPUT` list in `configure.ac`.

## 2.3.7. The Core/tool Interface

The core/tool interface evolves over time, but it's pretty stable. We deliberately do not provide backward compatibility with old interfaces, because it is too difficult and too restrictive. We view this as a good thing -- if we had to be backward compatible with earlier versions, many improvements now in the system could not have been added.

Because tools are statically linked with the core, if a tool compiles successfully then it should be compatible with the core. We would not deliberately violate this property by, for example, changing the behaviour of a core function without changing its prototype.

## 2.4. Final Words

Writing a new Valgrind tool is not easy, but the tools you can write with Valgrind are among the most powerful programming tools there are. Happy programming!

# 3. Callgrind Format Specification

This chapter describes the Callgrind Format, Version 1.

The format description is meant for the user to be able to understand the file contents; but more important, it is given for authors of measurement or visualization tools to be able to write and read this format.

## 3.1. Overview

The profile data format is ASCII based. It is written by Callgrind, and it is upwards compatible to the format used by Cachegrind (ie. Cachegrind uses a subset). It can be read by `callgrind_annotate` and `KCachegrind`.

This chapter gives an overview of format features and examples. For detailed syntax, look at the format reference.

### 3.1.1. Basic Structure

To uniquely specify that a file is a callgrind profile, it should add `"# callgrind format"` as first line. This is optional but recommended for easy format detection.

Each file has a header part of an arbitrary number of lines of the format `"key: value"`. After the header, lines specifying profile costs follow. Everywhere, comments on own lines starting with `'#'` are allowed. The header lines with keys `"positions"` and `"events"` define the meaning of cost lines in the second part of the file: the value of `"positions"` is a list of subpositions, and the value of `"events"` is a list of event type names. Cost lines consist of subpositions followed by 64-bit counters for the events, in the order specified by the `"positions"` and `"events"` header line.

The `"events"` header line is always required in contrast to the optional line for `"positions"`, which defaults to `"line"`, i.e. a line number of some source file. In addition, the second part of the file contains position specifications of the form `"spec=name"`. `"spec"` can be e.g. `"fn"` for a function name or `"fl"` for a file name. Cost lines are always related to the function/file specifications given directly before.

### 3.1.2. Simple Example

The event names in the following example are quite arbitrary, and are not related to event names used by Callgrind. Especially, cycle counts matching real processors probably will never be generated by any Valgrind tools, as these are bound to simulations of simple machine models for acceptable slowdown. However, any profiling tool could use the format described in this chapter.

```
# callgrind format
events: Cycles Instructions Flops
fl=file.f
fn=main
15 90 14 2
16 20 12
```

The above example gives profile information for event types `"Cycles"`, `"Instructions"`, and `"Flops"`. Thus, cost lines give the number of CPU cycles passed by, number of executed instructions, and number of floating point operations executed while running code corresponding to some source position. As there is no line specifying the value of `"positions"`, it defaults to `"line"`, which means that the first number of a cost line is always a line number.

Thus, the first cost line specifies that in line 15 of source file `file.f` there is code belonging to function `main`. While running, 90 CPU cycles passed by, and 2 of the 14 instructions executed were floating point operations. Similarly, the next line specifies that there were 12 instructions executed in the context of function `main` which can be related to line 16 in file `file.f`, taking 20 CPU cycles. If a cost line specifies less event counts than given in the `"events"` line, the rest is assumed to be zero. I.e. there was no floating point instruction executed relating to line 16.

Note that regular cost lines always give self (also called exclusive) cost of code at a given position. If you specify multiple cost lines for the same position, these will be summed up. On the other hand, in the example above there is no specification of how many times function `main` actually was called: profile data only contains sums.

### 3.1.3. Associations

The most important extension to the original format of Cachegrind is the ability to specify call relationship among functions. More generally, you specify associations among positions. For this, the second part of the file also can contain association specifications. These look similar to position specifications, but consist of two lines. For calls, the format looks like

```
calls=(Call Count) (Target position)
(Source position) (Inclusive cost of call)
```

The destination only specifies subpositions like line number. Therefore, to be able to specify a call to another function in another source file, you have to precede the above lines with a `"cfn="` specification for the name of the called function, and optionally a `"cfi="` specification if the function is in another source file (`"cfl="` is an alternative specification for `"cfi="` because of historical reasons, and both should be supported by format readers). The second line looks like a regular cost line with the difference that inclusive cost spent inside of the function call has to be specified.

Other associations are for example (conditional) jumps. See the reference below for details.

### 3.1.4. Extended Example

The following example shows 3 functions, `main`, `func1`, and `func2`. Function `main` calls `func1` once and `func2` 3 times. `func1` calls `func2` 2 times.

```
# callgrind format
events: Instructions

fl=file1.c
fn=main
16 20
cfn=func1
calls=1 50
16 400
cfi=file2.c
cfn=func2
calls=3 20
16 400

fn=func1
51 100
cfi=file2.c
cfn=func2
calls=2 20
51 300

fl=file2.c
fn=func2
20 700
```

One can see that in `main` only code from line 16 is executed where also the other functions are called. Inclusive cost of `main` is 820, which is the sum of self cost 20 and costs spent in the calls: 400 for the single call to `func1` and 400 as sum for the three calls to `func2`.

Function `func1` is located in `file1.c`, the same as `main`. Therefore, a `"cfi="` specification for the call to `func1` is not needed. The function `func1` only consists of code at line 51 of `file1.c`, where `func2` is called.

### 3.1.5. Name Compression

With the introduction of association specifications like calls it is needed to specify the same function or same file name multiple times. As absolute filenames or symbol names in C++ can be quite long, it is advantageous to be able to specify integer IDs for position specifications. Here, the term "position" corresponds to a file name (source or object file) or function name.

To support name compression, a position specification can be not only of the format `"spec=name"`, but also `"spec=(ID) name"` to specify a mapping of an integer ID to a name, and `"spec=(ID)"` to reference a previously defined ID mapping. There is a separate ID mapping for each position specification, i.e. you can use ID 1 for both a file name and a symbol name.

With string compression, the example from above looks like this:

```
# callgrind format
events: Instructions

fl=(1) file1.c
fn=(1) main
16 20
cfn=(2) func1
calls=1 50
16 400
cfi=(2) file2.c
cfn=(3) func2
calls=3 20
16 400

fn=(2)
51 100
cfi=(2)
cfn=(3)
calls=2 20
51 300

fl=(2)
fn=(3)
20 700
```

As position specifications carry no information themselves, but only change the meaning of subsequent cost lines or associations, they can appear everywhere in the file without any negative consequence. Especially, you can define name compression mappings directly after the header, and before any cost lines. Thus, the above example can also be written as

```
# callgrind format
events: Instructions

# define file ID mapping
fl=(1) file1.c
fl=(2) file2.c
# define function ID mapping
fn=(1) main
fn=(2) func1
fn=(3) func2

fl=(1)
```

```
fn=(1)
16 20
...
```

### 3.1.6. Subposition Compression

If a Callgrind data file should hold costs for each assembler instruction of a program, you specify subposition "instr" in the "positions:" header line, and each cost line has to include the address of some instruction. Addresses are allowed to have a size of 64 bits to support 64-bit architectures. Thus, repeating similar, long addresses for almost every line in the data file can enlarge the file size quite significantly, and motivates for subposition compression: instead of every cost line starting with a 16 character long address, one is allowed to specify relative addresses. This relative specification is not only allowed for instruction addresses, but also for line numbers; both addresses and line numbers are called "subpositions".

A relative subposition always is based on the corresponding subposition of the last cost line, and starts with a "+" to specify a positive difference, a "-" to specify a negative difference, or consists of "\*" to specify the same subposition. Because absolute subpositions always are positive (ie. never prefixed by "-"), any relative specification is non-ambiguous; additionally, absolute and relative subposition specifications can be mixed freely. Assume the following example (subpositions can always be specified as hexadecimal numbers, beginning with "0x"):

```
# callgrind format
positions: instr line
events: ticks

fn=func
0x80001234 90 1
0x80001237 90 5
0x80001238 91 6
```

With subposition compression, this looks like

```
# callgrind format
positions: instr line
events: ticks

fn=func
0x80001234 90 1
+3 * 5
+1 +1 6
```

Remark: For assembler annotation to work, instruction addresses have to be corrected to correspond to addresses found in the original binary. I.e. for relocatable shared objects, often a load offset has to be subtracted.

### 3.1.7. Miscellaneous

#### 3.1.7.1. Cost Summary Information

For the visualization to be able to show cost percentage, a sum of the cost of the full run has to be known. Usually, it is assumed that this is the sum of all cost lines in a file. But sometimes, this is not correct. Thus, you can specify a "summary:" line in the header giving the full cost for the profile run. An import filter may use this to show a progress bar while loading a large data file.

#### 3.1.7.2. Long Names for Event Types and inherited Types

Event types for cost lines are specified in the "events:" line with an abbreviated name. For visualization, it makes sense to be able to specify some longer, more descriptive name. For an event type "Ir" which means "Instruction Fetches", this can be specified the header line

```
event: Ir : Instruction Fetches
events: Ir Dr
```

In this example, "Dr" itself has no long name associated. The order of "event:" lines and the "events:" line is of no importance. Additionally, inherited event types can be introduced for which no raw data is available, but which are calculated from given types. Suppose the last example, you could add

```
event: Sum = Ir + Dr
```

to specify an additional event type "Sum", which is calculated by adding costs for "Ir and "Dr".

## 3.2. Reference

### 3.2.1. Grammar

```
ProfileDataFile := FormatSpec? FormatVersion? Creator? PartData*
```

```
FormatSpec := "# callgrind format\n"
```

```
FormatVersion := "version: 1\n"
```

```
Creator := "creator:" NoNewLineChar* "\n"
```

```
PartData := (HeaderLine "\n")+ (BodyLine "\n")+
```

```
HeaderLine := (empty line)
```

```
| ('#' NoNewLineChar*)
| PartDetail
| Description
| EventSpecification
| CostLineDef
```

```
PartDetail := TargetCommand | TargetID
```

```
TargetCommand := "cmd:" Space* NoNewLineChar*
```

```
TargetID := ("pid"|"thread"|"part") ":" Space* Number
```

```
Description := "desc:" Space* Name Space* ":" NoNewLineChar*
```

```
EventSpecification := "event:" Space* Name InheritedDef? LongNameDef?
```

```
InheritedDef := "=" InheritedExpr
```

```
InheritedExpr := Name
```

```
| Number Space* ("*" Space*)? Name
| InheritedExpr Space* "+" Space* InheritedExpr
```

```
LongNameDef := ":" NoNewLineChar*
```

```
CostLineDef := "events:" Space* Name (Space+ Name)*
| "positions:" "instr"? (Space+ "line")?
```

```
BodyLine := (empty line)
```

```
| ('#' NoNewLineChar*)
| CostLine
| PositionSpec
| CallSpec
| UncondJumpSpec
```

	CondJumpSpec
CostLine	:= SubPositionList Costs?
SubPositionList	:= (SubPosition+ Space+)+
SubPosition	:= Number   "+" Number   "-" Number   "*"   "/"
Costs	:= (Number Space+)+
PositionSpec	:= Position "=" Space* PositionName
Position	:= CostPosition   CalledPosition
CostPosition	:= "ob"   "fl"   "fi"   "fe"   "fn"
CalledPosition	:= "cob"   "cfl"   "cfi"   "cfn"
PositionName	:= ( "(" Number ")" )? (Space* NoNewLineChar* )?
CallSpec	:= CallLine "\n" CostLine
CallLine	:= "calls=" Space* Number Space+ SubPositionList
UncondJumpSpec	:= "jump=" Space* Number Space+ SubPositionList
CondJumpSpec	:= "jcond=" Space* Number Space+ Number Space+ SubPositionList
Space	:= " "   "\t"
Number	:= HexNumber   (Digit)+
Digit	:= "0"   ...   "9"
HexNumber	:= "0x" (Digit   HexChar)+
HexChar	:= "a"   ...   "f"   "A"   ...   "F"
Name	= Alpha (Digit   Alpha)*
Alpha	= "a"   ...   "z"   "A"   ...   "Z"
NoNewLineChar	:= all characters without "\n"

A profile data file ("ProfileDataFile") starts with basic information such as a format marker, the version and creator information, and then has a list of parts, where each part has its own header and body. Parts typically are different threads and/or time spans/phases within a profiled application run.

Note that callgrind\_annotate currently only supports profile data files with one part. Callgrind may produce multiple parts for one profile run, but defaults to one output file for each part.

## 3.2.2. Description of Header Lines

Basic information in the first lines of a profile data file:

- # callgrind format [Callgrind]

This line specifies that the file is a callgrind profile, and it has to be the first line. It was added late to the format (with Valgrind 3.13) and is optional, as all readers also should work with older callgrind profiles not including this line. However, generation of this line is recommended to allow desktop environments and file managers to uniquely detect the format.

- `version: number` [Callgrind]

This is used to distinguish future profile data formats. A major version of 0 or 1 is supposed to be upwards compatible with Cachegrind's format. It is optional; if not appearing, version 1 is assumed. Otherwise, it has to follow directly after the format specification (i.e. be the first line if the optional format specification is skipped).

- `creator: string` [Callgrind]

This is an arbitrary string to denote the creator of this file. Optional.

The header for each part has an arbitrary number of lines of the format "key: value". Possible *key* values for the header are:

- `pid: process id` [Callgrind]

Optional. This specifies the process ID of the supervised application for which this profile was generated.

- `cmd: program name + args` [Cachegrind]

Optional. This specifies the full command line of the supervised application for which this profile was generated.

- `part: number` [Callgrind]

Optional. This specifies a sequentially incremented number for each dump generated, starting at 1.

- `desc: type: value` [Cachegrind]

This specifies various information for this dump. For some types, the semantic is defined, but any description type is allowed. Unknown types should be ignored.

There are the types "I1 cache", "D1 cache", "LL cache", which specify parameters used for the cache simulator. These are the only types originally used by Cachegrind. Additionally, Callgrind uses the following types: "Timerange" gives a rough range of the basic block counter, for which the cost of this dump was collected. Type "Trigger" states the reason of why this trace was generated. E.g. program termination or forced interactive dump.

- `positions: [instr] [line]` [Callgrind]

For cost lines, this defines the semantic of the first numbers. Any combination of "instr", "bb" and "line" is allowed, but has to be in this order which corresponds to position numbers at the start of the cost lines later in the file.

If "instr" is specified, the position is the address of an instruction whose execution raised the events given later on the line. This address is relative to the offset of the binary/shared library file to not have to specify relocation info. For "line", the position is the line number of a source file, which is responsible for the events raised. Note that the mapping of "instr" and "line" positions are given by the debugging line information produced by the compiler.

This header line is optional, defaulting to "positions: line" if not specified.

- `events: event type abbreviations` [Cachegrind]

A list of short names of the event types logged in cost lines in this part of the profile data file. Arbitrary short names are allowed. The order given specifies the required order in cost lines. Thus, the first event type is the second or third number in a cost line, depending on the value of "positions". Required to appear for each header part exactly once.

- `summary: costs` [Callgrind]

Optional. This header line specifies a summary cost, which should be equal or larger than a total over all self costs. It may be larger as the cost lines may not represent all cost of the program run.

- `totals: costs` [Cachegrind]

Optional. Should appear at the end of the file (although looking like a header line). Must give the total of all cost lines, to allow for a consistency check.

### 3.2.3. Description of Body Lines

The regular body line is a cost line consisting of one or two position numbers (depending on "positions:" header line, see above) and an array of cost numbers. A position number either is a line numbers into a source file or an instruction address within binary code, with source/binary file names specified as position names (see below). The cost numbers get mapped to event types in the same order as specified in the "events:" header line. If less numbers than event types are given, the costs default to zero for the remaining event types.

Further, there exist lines `spec=position name`. A position name is an arbitrary string. If it starts with "(" and a digit, it's a string in compressed format. Otherwise it's the real position string. This allows for file and symbol names as position strings, as these never start with "(" + *digit*. The compressed format is either "(" *number* ")" *space position* or only "(" *number* ")". The first relates *position* to *number* in the context of the given format specification from this line to the end of the file; it makes the (*number*) an alias for *position*. Compressed format is always optional.

Position specifications allowed:

- `ob= [Callgrind]`

The ELF object where the cost of next cost lines happens.

- `fl= [Cachegrind]`
- `fi= [Cachegrind]`
- `fe= [Cachegrind]`

The source file including the code which is responsible for the cost of next cost lines. "fi="/"fe=" is used when the source file changes inside of a function, i.e. for inlined code.

- `fn= [Cachegrind]`

The name of the function where the cost of next cost lines happens.

- `cob= [Callgrind]`

The ELF object of the target of the next call cost lines.

- `cfi= [Callgrind]`

The source file including the code of the target of the next call cost lines.

- `cfl= [Callgrind]`

Alternative spelling for `cfi=` specification (because of historical reasons).

- `cfn= [Callgrind]`

The name of the target function of the next call cost lines.

The last type of body line provides specific costs not just related to one position as regular cost lines. It starts with specific strings similar to position name specifications.

- `calls=count target-position [Callgrind]`

Call executed "count" times to "target-position". After a "calls=" line there **MUST** be a cost line. This provides the source position of the call and the cost spent in the called function in total.

- `jump=count target-position [Callgrind]`

Unconditional jump, executed "count" times, to "target-position".

- `jcnd=exe-count jump-count target-position [Callgrind]`

Conditional jump, executed "exe-count" times with "jump-count" jumps happening (rest is fall-through) to "target-position".

# Valgrind Distribution Documents

**Release 3.23.0.GIT ?? Apr 2024**

**Copyright © 2000-2022 [Valgrind Developers](#)**

Email: [valgrind@valgrind.org](mailto:valgrind@valgrind.org)

## Table of Contents

1. AUTHORS .....	1
2. NEWS .....	3
3. OLDER NEWS .....	13
4. README .....	71
5. README_MISSING_SYSCALL_OR_IOCTL .....	73
6. README_DEVELOPERS .....	78
7. README_PACKAGERS .....	85
8. README.S390 .....	88
9. README.android .....	89
10. README.android_emulator .....	93
11. README.mips .....	95
12. README.solaris .....	97
13. README.freebsd .....	101

# 1. AUTHORS

Julian Seward was the original founder, designer and author of Valgrind, created the dynamic translation frameworks, wrote Memcheck, the 3.X versions of Helgrind, SGCheck, DHAT, and did lots of other things.

Nicholas Nethercote did the core/tool generalisation, wrote Cachegrind and Massif, and tons of other stuff.

Tom Hughes did a vast number of bug fixes, helped out with support for more recent Linux/glibc versions, set up the present build system, and has helped out with test and build machines.

Jeremy Fitzhardinge wrote Helgrind (in the 2.X line) and totally overhauled low-level syscall/signal and address space layout stuff, among many other things.

Josef Weidendorfer wrote and maintains Callgrind and the associated KCachegrind GUI.

Paul Mackerras did a lot of the initial per-architecture factoring that forms the basis of the 3.0 line and was also seen in 2.4.0. He also did UCode-based dynamic translation support for PowerPC, and created a set of ppc-linux derivatives of the 2.X release line.

Greg Parker wrote the Mac OS X port.

Dirk Mueller contributed the malloc/free mismatch checking and other bits and pieces, and acts as our KDE liaison.

Robert Walsh added file descriptor leakage checking, new library interception machinery, support for client allocation pools, and minor other tweakage.

Bart Van Assche wrote and maintains DRD.

Cerion Armour-Brown worked on PowerPC instruction set support in the Vex dynamic-translation framework. Maynard Johnson improved the Power6 support.

Kirill Batuzov and Dmitry Zhurikhin did the NEON instruction set support for ARM. Donna Robinson did the v6 media instruction support.

Donna Robinson created and maintains the very excellent <http://www.valgrind.org>.

Vince Weaver wrote and maintains BBV.

Frederic Gobry helped with autoconf and automake.

Daniel Berlin modified readelf's dwarf2 source line reader, written by Nick Clifton, for use in Valgrind.

Michael Matz and Simon Hausmann modified the GNU binutils demangler(s) for

use in Valgrind.

David Woodhouse has helped out with test and build machines over the course of many releases.

Florian Krohm and Christian Borntraeger wrote the initial S390X/Linux port. Andreas Arnez is the current maintainer and developer of it. Florian improved and ruggedised the regression test system during 2011.

Philippe Waroquiers wrote and maintains the embedded GDB server. He also made a bunch of performance and memory-reduction fixes across diverse parts of the system.

Carl Love and Maynard Johnson contributed IBM Power6 and Power7 support, and generally deal with ppc{32,64}-linux issues.

Petar Jovanovic and Dejan Jevtic wrote and maintain the mips32-linux port.

Dragos Tatulea modified the arm-android port so it also works on x86-android.

Jakub Jelinek helped out extensively with the AVX and AVX2 support.

Mark Wielaard fixed a bunch of bugs and acts as our Fedora/RHEL liaison.

Assad Hashmi contributed support for AArch64 v8.1 and later.

Maran Pakkirisamy implemented support for decimal floating point on s390.

Rhys Kidd updated and maintains the macOS port.

Paul Floyd maintains the FreeBSD port and occasionally fixes Solaris and macOS issues.

Many, many people sent bug reports, patches, and helpful feedback.

Development of Valgrind was supported in part by the Tri-Lab Partners (Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratories) of the U.S. Department of Energy's Advanced Simulation & Computing (ASC) Program.

## 2. NEWS

Release 3.23.0 (?? Apr 2024)

~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/macOSX 10.12, X86/FreeBSD and AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

\* ===== CORE CHANGES =====

\* --track-fds=yes will now also warn about double closing of file descriptors. Printing the context where the file descriptor was originally opened and where it was previously closed.

\* ===== PLATFORM CHANGES =====

\* ===== TOOL CHANGES =====

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

283429 ARM leak checking needs CLEAR\_CALLER\_SAVED\_REGS  
281059 Cannot connect to Oracle using valgrind  
369723 \_\_builtin\_longjmp not supported in clang/llvm on Android arm64 target  
390269 unhandled amd64-darwin syscall: unix:464 (openat\_nocancel)  
401284 False positive "Source and destination overlap in strncat"  
428364 Signals inside io\_uring\_enter not handled  
437790 valgrind reports "Conditional jump or move depends on uninitialised value" in memchr of macOS 10.12-10.15  
460616 disInstr(arm64): unhandled instruction 0x4E819402 (dotprod/ASIMDDP)  
466762 Add redirs for C23 free\_sized() and free\_aligned\_sized()  
466884 Missing writev uninit padding suppression for \_XSend  
471036 disInstr\_AMD64: disInstr miscalculated next %rip on RORX imm8, m32/64, r32/6  
471222 support tracking of file descriptors being double closed  
474160 If errors-for-leak-kinds is specified, exit-on-first-error should only exit on one of the listed errors.  
475498 Add reallocarray wrapper  
476320 Build failure with GCC  
476331 clean up generated/distributed filter scripts  
476535 Difference in allocation size for massif/tests/overloaded-new between clang++/libc++ and g++/libstdc++  
476548 valgrind 3.22.0 fails on assertion when loading debuginfo file produced by mold  
476708 valgrind-monitor.py regular expressions should use raw strings  
476780 Extend strlcat and strlcpy wrappers to GNU libc

476787 Build of Valgrind 3.21.0 fails when SOLARIS\_PT\_SUNDWTRACE\_THRP is defined  
 476887 WARNING: unhandled amd64-freebsd syscall: 578  
 477198 Add fchmodat2 syscall on linux  
 477628 Add mmap support for Solaris  
 477630 Include ucontext.h rather than sys/ucontext.h in Solaris sources  
 477719 vgdb incorrectly replies to qRcmd packet  
 478211 Redundant code for vgdb.c and Valgrind core tools  
 478624 Valgrind incompatibility with binutils-2.42 on x86 with new nop patterns (unhandled instruction bytes: 0x2E 0x8D 0xB4 0x26)  
 478837 valgrind fails to read debug info for rust binaries  
 479041 Executables without RW sections do not trigger debuginfo reading  
 480052 WARNING: unhandled amd64-freebsd syscall: 580  
 480126 Build failure on Raspberry Pi 5 / OS 6.1.0-rpi7-rpi-v8  
 480405 valgrind 3.22.0 "m\_debuginfo/image.c:586 (set\_CEnt): Assertion '!sr\_isError(sr)' failed."  
 480488 Add support for FreeBSD 13.3  
 480706 Unhandled syscall 325 (mlock2)  
 481131 [PATCH] x86 regtest: fix clobber lists in generated asm statements  
 483786 Incorrect parameter indexing in FreeBSD clock\_nanosleep syscall wrapper  
 484002 Add suppression for invalid read in glibc's \_\_wcpncpy\_avx2() via wcsxfrm()  
 n-i-bz Add redirect for memccpy

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXXX)  
 where XXXXXX is the bug number as listed above.

(3.23.0.RC1: ?? Apr 2024)

Release 3.22.0 (31 Oct 2023)

~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

\* ===== CORE CHANGES =====

\* A new configure option --with-gdbscripts-dir lets you install the gdb valgrind python monitor scripts in a specific location. For example a distro could use it to install the scripts in a safe load location --with-gdbscripts-dir=%{\_datadir}/gdb/auto-load. It is also possible to configure --without-gdb-scripts-dir so no .debug\_gdb\_scripts section is added to the vgppreload library and no valgrind-monitor python scripts are installed at all.

\* ===== PLATFORM CHANGES =====

\* Support has been added for FreeBSD 14 and FreeBSD 15.  
 \* Add support for the following FreeBSD system calls: close\_range, kqueue, membarrier, timerfd\_create, timerfd\_settime and timerfd\_gettime (all added in FreeBSD 15).

\* ===== TOOL CHANGES =====

\* Memcheck now tests and warns about the values used for alignment and size. These apply to various functions: memalign, posix\_memalign and aligned\_alloc in C and various overloads of operators new and delete in C++. The kinds of error that can be detected are

- invalid alignment, for instance the alignment is usually required to be a power of 2
- mismatched alignment between aligned allocation and aligned deallocation
- mismatched size when sized delete is used
- bad size for functions that have implementation defined behaviour when the requested size is zero

\* Cachegrind:

- You can now profile part of a program's execution using the new ``CACHEGRIND_START_INSTRUMENTATION`` and ``CACHEGRIND_STOP_INSTRUMENTATION`` client requests, along with the new ``--instr-at-start`` option. The behaviour is the same as Callgrind's equivalent functionality.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

390871 ELF debug info reader confused with multiple .rodata\* sections  
 417993 vbit-test fail on s390x with Iop\_Add32: spurious dependency on uninit  
 426751 Valgrind reports "still reachable" memory using musl  
     (alpine running inside docker)  
 432801 Valgrind 3.16.1 reports a jump based on uninitialized memory somehow  
     related to clang and signals  
 433857 Add validation to C++17 aligned new/delete alignment size  
 433859 Add mismatched detection to C++ 17 aligned new/delete  
 460192 Add epoll\_pwait2  
 461074 DWARF2 CFI reader: unhandled DW\_OP\_0x11 (consts) DW\_OP\_0x92 (bregx)  
 465782 s390x: Valgrind doesn't compile with Clang on s390x  
 466105 aligned\_alloc problems, part 2  
 467441 Add mismatched detection to C++ 14 sized delete  
 469049 link failure on ppc64 (big endian) valgrind 3.20  
 469146 massif --ignore-fn does not ignore inlined functions  
 469768 Make it possible to install gdb scripts in a different location  
 470121 Can't run callgrind\_control with valgrind 3.21.0 because of perl errors  
 470132 s390x: Assertion failure on VGM instruction  
 470520 Multiple realloc zero errors crash in MC\_(eq\_Error)  
 470713 Failure on the Yosys project: valgrind: m\_libcfile.c:1802  
     (Bool vgPlain\_realpath(const HChar \*, HChar \*)):  
     Assertion 'resolved' failed  
 470830 Don't print actions vgdb me ... continue for vgdb --multi mode  
 470978 s390x: Valgrind cannot start qemu-kvm when "sysctl vm.allocate\_pgste=0"  
 471311 gdb --multi mode stdout redirecting to stderr  
 471807 Add support for lazy reading and downloading of DWARF debuginfo  
 472219 Syscall param ppoll(ufds.events) points to uninitialised byte(s)  
 472875 none/tests/s390x/dfp-1 failure  
 472963 Broken regular expression in configure.ac  
 473604 Fix bug472219.c compile failure with Clang 16

473677 make check compile failure with Clang 16 based on GCC 13.x  
 473745 must-be-redirection function - strlen  
 473870 FreeBSD 14 applications fail early at startup  
 473944 Handle mold linker split RW PT\_LOAD segments correctly  
 474332 aligned\_alloc under Valgrind returns nullptr when alignment is not a multiple of sizeof(void \*)  
 475650 DRD does not work with C11 threads  
 475652 Missing suppression for \_\_wcsncpy\_avx2 (strncpy-avx2.S:308)?  
 476108 vg\_replace\_malloc DELETE checks size  
 n-i-bz Allow arguments with spaces in .valgrindrc files  
 n-i-bz FreeBSD fixed reading of Valgrind tools own debuginfo

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed above.

(3.22.0.RC1: 17 Oct 2023)

(3.22.0.RC2: 26 Oct 2023)

Release 3.21.0 (28 Apr 2023)

~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,  
 PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,  
 MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,  
 X86/Solaris, AMD64/Solaris, AMD64/macOSX 10.12, X86/FreeBSD and  
 AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13,  
 AMD64/macOS 10.13 and nanoMIPS/Linux.

\* ===== CORE CHANGES =====

\* When GDB is used to debug a program running under valgrind using  
 the valgrind gdbserver, GDB will automatically load some  
 python code provided in valgrind defining GDB front end commands  
 corresponding to the valgrind monitor commands.

These GDB front end commands accept the same format as  
 the monitor commands directly sent to the Valgrind gdbserver.

These GDB front end commands provide a better integration  
 in the GDB command line interface, so as to use for example  
 GDB auto-completion, command specific help, searching for  
 a command or command help matching a regexp, ...

For relevant monitor commands, GDB will evaluate arguments  
 to make the use of monitor commands easier.

For example, instead of having to print the address of a variable  
 to pass it to a subsequent monitor command, the GDB front end  
 command will evaluate the address argument. It is for example  
 possible to do:

(gdb) memcheck who\_points\_at &some\_struct sizeof(some\_struct)

instead of:

(gdb) p &some\_struct

\$2 = (some\_struct\_type \*) 0x1130a0 <some\_struct>

(gdb) p sizeof(some\_struct)

\$3 = 40

(gdb) monitor who\_point\_at 0x1130a0 40

\* The vgdb utility now supports extended-remote protocol when  
 invoked with --multi. In this mode the GDB run command is  
 supported. Which means you don't need to run gdb and valgrind  
 from different terminals. So for example to start your program

in gdb and run it under valgrind you can do:

```
$ gdb prog
(gdb) set remote exec-file prog
(gdb) set sysroot /
(gdb) target extended-remote | vgdb --multi
(gdb) start
```

\* The behaviour of realloc with a size of zero can now be changed for tools that intercept malloc. Those tools are memcheck, helgrind, drd, massif and dhat. Realloc implementations generally do one of two things

- free the memory like free() and return NULL (GNU libc and ptmalloc).
- either free the memory and then allocate a minimum sized block or just return the original pointer. Return NULL if the allocation of the minimum sized block fails (jemalloc, musl, snmalloc, Solaris, macOS).

When Valgrind is configured and built it will try to match the OS and libc behaviour. However if you are using a non-default library to replace malloc and family (e.g., musl on a glibc Linux or tcmalloc on FreeBSD) then you can use a command line option to change the behaviour of Valgrind:

```
--realloc-zero-bytes-frees=yes|no [yes on Linux glibc, no otherwise]
```

\* ===== PLATFORM CHANGES =====

\* Make the address space limit on FreeBSD amd64 128Gbytes (the same as Linux and Solaris, it was 32Gbytes)

\* ===== TOOL CHANGES =====

\* Memcheck:

- When doing a delta leak\_search, it is now possible to only output the new loss records compared to the previous leak search. This is available in the memcheck monitor command 'leak\_search' by specifying the "new" keyword or in your program by using the client request VALGRIND\_DO\_NEW\_LEAK\_CHECK. Whenever a "delta" leak search is done (i.e. when specifying "new" or "increased" or "changed" in the monitor command), the new loss records have a "new" marker.
- Valgrind now contains python code that defines GDB memcheck front end monitor commands. See CORE CHANGES.
- Performs checks for the use of realloc with a size of zero. This is non-portable and a source of errors. If memcheck detects such a usage it will generate an error  
realloc() with size 0  
followed by the usual callstacks.  
A switch has been added to allow this to be turned off:  
--show-realloc-size-zero=yes|no [yes]

\* Helgrind:

- The option ---history-backtrace-size=<number> allows to configure the number of entries to record in the stack traces of "old" accesses. Previously, this number was hardcoded to 8.
- Valgrind now contains python code that defines GDB helgrind front end monitor commands. See CORE CHANGES.

\* Cachegrind:

- `--cache-sim=no` is now the default. The cache simulation is old and unlikely to match any real modern machine. This means only the `Ir` event are gathered by default, but that is by far the most useful event.
- `cg_annotate`, `cg_diff`, and `cg_merge` have been rewritten in Python. As a result, they all have more flexible command line argument handling, e.g. supporting `--show-percs` and `--no-show-percs` forms as well as the existing `--show-percs=yes` and `--show-percs=no`.
- `cg_annotate` has some functional changes.
  - It's much faster, e.g. 3-4x on common cases.
  - It now supports diffing (with `--diff`, `--mod-filename`, and `--mod-funcname`) and merging (by passing multiple data files).
  - It now provides more information at the file and function level. There are now "File:function" and "Function:file" sections. These are very useful for programs that use inlining a lot.
  - Support for user-annotated files and the `-I/--include` option has been removed, because it was of little use and blocked other improvements.
  - The `--auto` option is renamed `--annotate`, though the old `--auto=yes`/`--auto=no` forms are still supported.
- `cg_diff` and `cg_merge` are now deprecated, because `cg_annotate` now does a better job of diffing and merging.
- The Cachegrind output file format has changed very slightly, but in ways nobody is likely to notice.

\* Callgrind:

- Valgrind now contains python code that defines GDB callgrind front end monitor commands. See CORE CHANGES.

\* Massif:

- Valgrind now contains python code that defines GDB massif front end monitor commands. See CORE CHANGES.

\* DHAT:

- A new kind of user request has been added which allows you to override the 1024 byte limit on access count histograms for blocks of memory. The client request is `DHAT_HISTOGRAM_MEMORY`.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

170510 Don't warn about ioctl of size 0 without direction hint  
 241072 List tools in --help output  
 327548 false positive while destroying mutex  
 382034 Testcases build fixes for musl  
 351857 confusing error message about valid command line option  
 374596 inconsistent RDTSCP support on x86\_64  
 392331 Spurious lock not held error from inside pthread\_cond\_timedwait  
 397083 Likely false positive "uninitialised value(s)" for `__wmemchr_avx2` and `__wmemcmp_avx2_movbe`

400793 pthread\_rwlock\_timedwrlock false positive  
 419054 Unhandled syscall getcpu on arm32  
 433873 openat2 syscall unimplemented on Linux  
 434057 Add stdio mode to valgrind's gdbserver  
 435441 valgrind fails to interpose malloc on musl 1.2.2 due to weak symbol name and no libc soname  
 436413 Warn about realloc of size zero  
 439685 compiler warning in callgrind/main.c  
 444110 priv/guest\_ppc\_toIR.c:36198:31: warning: duplicated 'if' condition.  
 444487 hginfo test detects an extra lock inside data symbol "\_rtld\_local"  
 444488 Use glibc.pthread.stack\_cache\_size tunable  
 444568 drd/tests/pth\_barrier\_thr\_cr fails on Fedora 38  
 445743 "The impossible happened: mutex is locked simultaneously by two threads"  
     while using mutexes with priority inheritance and signals  
 449309 Missing loopback device ioctl(s)  
 459476 vgdb: allow address reuse to avoid "address already in use" errorsuse" errors  
 460356 s390: Sqrt32Fx4 -- cannot reduce tree  
 462830 WARNING: unhandled amd64-freebsd syscall: 474  
 463027 broken check for MPX instruction support in assembler  
 464103 Enhancement: add a client request to DHAT to mark memory to be histogrammed  
 464476 Firefox fails to start under Valgrind  
 464609 Valgrind memcheck should support Linux pidfd\_open  
 464680 Show issues caused by memory policies like selinux deny\_execmem  
 464859 Build failures with GCC-13 (drd tsan\_unittest)  
 464969 D language demangling  
 465435 m\_libcfile.c:66 (vgPlain\_safe\_fd): Assertion 'newfd >= VG\_(fd\_hard\_limit)' failed.  
 466104 aligned\_alloc problems, part 1  
 467036 Add time cost statistics for Regtest  
 467482 Build failure on aarch64 Alpine  
 467714 fdleak\_\* and rlimit tests fail when parent process has more than  
     64 descriptors opened  
 467839 Gdbserver: Improve compatibility of library directory name  
 468401 [PATCH] Add a style file for clang-format  
 468556 Build failure for vgdb  
 468606 build: remove "Valgrind relies on GCC" check/output  
 469097 ppc64(be) doesn't support SCV syscall instruction  
 n-i-bz FreeBSD rfork syscall fail with EINVAL or ENOSYS rather than VG\_(unimplemented)

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed above.

\* ===== KNOWN ISSUES =====

\* configure --enable-lto=yes is known to not work in all setups.  
 See bug 469049. Workaround: Build without LTO.

(3.21.0.RC1: 14 Apr 2023)

(3.21.0.RC2: 21 Apr 2023)

Release 3.20.0 (24 Oct 2022)

~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux,  
 PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux,  
 MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,  
 X86/Solaris, AMD64/Solaris, AMD64/macOSX 10.12, X86/FreeBSD and  
 AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13,  
 AMD64/macOS 10.13 and nanoMIPS/Linux.

\* ===== CORE CHANGES =====

- \* The option "--vgdb-stop-at=event1,event2,..." accepts the new value abexit. This indicates to invoke gdbserver when your program exits abnormally (i.e. with a non zero exit code).
- \* Fix Rust v0 name demangling.
- \* The Linux rseq syscall is now implemented as (silently) returning ENOSYS.
- \* Add FreeBSD syscall wrappers for \_\_specialfd and \_\_realpathat.
- \* Remove FreeBSD dependencies on COMPAT10, which fixes compatibility with HardenedBSD
- \* The option --enable-debuginfod=<no|yes> [default: yes] has been added on Linux.
- \* More DWARF5 support as generated by clang14.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

131186 writev reports error in (vector[...])  
 434764 iconv\_open causes ld.so v2.28+ to use optimised strncmp  
 446754 Improve error codes from alloc functions under memcheck  
 452274 memcheck crashes with Assertion 'sci->status.what == SsIdle' failed  
 452779 Valgrind fails to build on FreeBSD 13.0 with llvm-devel (15.0.0)  
 453055 shared\_timed\_mutex drd test fails with "Lock shared failed" message  
 453602 Missing command line option to enable/disable debuginfod  
 452802 Handle lld 9+ split RW PT\_LOAD segments correctly  
 454040 s390x: False-positive memcheck:cond in memmem on arch13 systems  
 456171 [PATCH] FreeBSD: Don't record address errors when accessing the 'kern.ps\_strings' sysctl struct  
 n-i-bz Implement vgdb invoker on FreeBSD  
 458845 PowerPC: The L field for the dcbf and sync instruction should be 3 bits in ISA 3.1.  
 458915 Remove register cache to fix 458915 gdbserver causes wrong syscall return  
 459031 Documentation on --error-exitcode incomplete  
 459477 XERROR messages lacks ending '\n' in vgdb  
 462007 Implicit int in none/tests/faultstatus.c

To see details of a given bug, visit  
[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed above.

(3.20.0.RC1: 20 Oct 2022)

Release 3.19.0 (11 Apr 2022)

~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

---

\* ===== CORE CHANGES =====

- \* Fix Rust v0 name demangling.
- \* The Linux rseq syscall is now implemented as (silently) returning ENOSYS.
- \* Add FreeBSD syscall wrappers for \_\_specialfd and \_\_realpathat.
- \* Remove FreeBSD dependencies on COMPAT10, which fixes compatibility with HardenedBSD

\* ===== PLATFORM CHANGES =====

- \* arm64:
  - ignore the "v8.x" architecture levels, only look at actual CPU features present. Fixes mismatch detected between RDMA and atomics features preventing startup on some QEMU configurations.
  - Implement LD{,A}XP and ST{,L}XP
  - Fix incorrect code emitted for doubleword CAS.
- \* s390:
  - Fix sys\_ipc semtimedop syscall
  - Fix VFLRX and WFLRX instructions
  - Fix EXRL instruction with negative offset
- \* ppc64:
  - Reimplement the vbpermq instruction support to generate less Iops and avoid overflowing internal buffers.
  - Fix checking for scv support to avoid "Facility 'SCV' unavailable (12), exception" messages in dmsg.
  - Fix setting condition code for Vector Compare quad word instructions.
  - Fix fix lxsibzx, lxsihzx and lxsihzx instructions so they only load their respective sized data.
  - Fix the prefixed stq instruction in PC relative mode.

\* ===== TOOL CHANGES =====

- \* Memcheck:
  - Speed up --track-origins=yes for large (in the range of hundreds to thousands of megabytes) mmap/munmaps.
- \* DRD/Helgrind:
  - Several fixes for new versions of libstd++ using new posix try\_lock functions

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

403802 leak\_cpp\_interior fails with some reachable blocks different than expected  
 435732 memcheck/tests/leak\_cpp\_interior fails with gcc11  
 444242 s390x: Valgrind crashes on EXRL with negative offset  
 444399 arm64: unhandled instruction 0xC87F2D89 (LD{,A}XP and ST{,L}XP).  
     == 434283  
 444481 gdb\_server test failures on s390x  
 444495 dhat/tests/copy fails on s390x  
 444552 memcheck/tests/sem fails on s390x with glibc 2.34

444571 PPC, fix the lxsibzx and lxsihzx so they only load their respective sized data.

444836 PPC, pstq instruction for R=1 is not storing to the correct address.

444925 fexecve syscall wrapper not properly implemented

445032 valgrind/memcheck crash with SIGSEGV when SIGVTALRM timer used and libthr.so associated

445211 Fix out of tree builds

445300 [PATCH] Fix building tests with Musl

445011 SIGCHLD is sent when valgrind uses debuginfod-find

445354 arm64 backend: incorrect code emitted for doubleword CAS

445415 arm64 front end: alignment checks missing for atomic instructions

445504 Using C++ condition\_variable results in bogus "mutex is locked simultaneously by two threads" warning

445607 Unhandled amd64-freebsd syscall: 247

445668 Inline stack frame generation is broken for Rust binaries

445916 Demangle Rust v0 symbols with .llvm suffix

446139 DRD/Helgrind with std::shared\_timed\_mutex::try\_lock\_until and try\_lock\_shared\_until false positives

446138 DRD/Helgrind with std::timed\_mutex::try\_lock\_until false positives

446281 Add a DRD suppression for fwrite

446103 Memcheck: `--track-origins=yes` causes extreme slowdowns for large mmap/munmap

446139 DRD/Helgrind with std::shared\_timed\_mutex::try\_lock\_until and try\_lock\_shared\_until false

446251 TARGET\_SIGNAL\_THR added to enum target\_signal

446823 FreeBSD - missing syscalls when using libzm4

447991 s390x: Valgrind indicates illegal instruction on wflrx

447995 Valgrind segfault on power10 due to hwcap checking code

449483 Powerpc: vcmpgtuq., vcmpgtuq., vcmpequq. instructions not setting the condition code correctly.

449672 ppc64 --track-origins=yes failures because of bad cmov addHRegUse

449838 sigsegv liburing the 'impossible' happened for io\_uring\_setup

450025 Powerc: ACC file not implemented as a logical overlay of the VSR registers.

450437 Warn for execve syscall with argv or argv[0] being NULL

450536 Powerpc: valgrind throws 'facility scv unavailable exception'

451626 Syscall param bpf(attr->raw\_tracepoint.name) points to unaddressable byte(s)

451827 [ppc64le] VEX temporary storage exhausted with several vbpermq instructions

451843 valgrind fails to start on a FreeBSD system which enforces W^X

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed above.

(3.19.0.RC1: 02 Apr 2022)

(3.19.0.RC2: 08 Apr 2022)

# 3. OLDER NEWS

Release 3.18.0 (15 Oct 2021)

~~~~~

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, AMD64/MacOSX 10.12, X86/FreeBSD and AMD64/FreeBSD. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

## \* ===== CORE CHANGES =====

- \* The libiberty demangler has been updated, which brings support for Rust v0 name demangling. [Update: alas, due to a bug, this support isn't working in 3.18.0.]
- \* `__libc_freeres` isn't called anymore after the program receives a fatal signal. Causing some internal glibc resources to hang around, but preventing any crashes after the program has ended.
- \* The DWARF reader is now very much faster at startup when just `--read-inline-info=yes` (the default in most cases) is given.
- \* glibc 2.34, which moved various functions from `libpthread.so` into `libc.so`, is now supported.

## \* ===== PLATFORM CHANGES =====

### \* arm64:

- v8.2 scalar and vector FABD, FACGE, FACGT and FADD.
- v8.2 FP compare & conditional compare instructions.
- Zero variants of v8.2 FP compare instructions.

### \* s390:

- Support the miscellaneous-instruction-extensions facility 3 and the vector-enhancements facility 2. This enables programs compiled with `"-march=arch13"` or `"-march=z15"` to be executed under Valgrind.

### \* ppc64:

- ISA 3.1 support is now complete
- ISA 3.0 support for the darn instruction added.
- ISA 3.0 support for the vector system call instruction `scv` added.
- ISA 3.0 support for the copy, paste and `cpabort` instructions added.

- \* Support for X86/FreeBSD and AMD64/FreeBSD has been added.

## \* ===== OTHER CHANGES =====

- \* Memcheck on amd64: minor fixes to remove some false positive undef-value errors

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

208531 [PATCH]: FreeBSD support for valgrind  
 368960 WARNING: unhandled amd64-linux syscall: 163 (acct)  
 407589 [Linux] Add support for C11 aligned\_alloc() and GNU reallocarray()  
 423963 Error in child thread when CLONE\_PIDFD is used  
 426148 crash with "impossible happened" when running BPF CO-RE programs  
 429375 PPC ISA 3.1 support is missing, part 9  
 431157 PPC\_FEATURE2\_SCV needs to be masked in AT\_HWCAP2  
 431306 Update demangler to support Rust v0 name mangling  
 432387 s390x: z15 instructions support  
 433437 FreeBSD support, part 1  
 433438 FreeBSD support, part 2  
 433439 FreeBSD support, part 3  
 433469 FreeBSD support, part 4  
 433473 FreeBSD support, part 5  
 433477 FreeBSD support, part 6  
 433479 FreeBSD support, part 7  
 433504 FreeBSD support, part 8  
 433506 FreeBSD support, part 9  
 433507 FreeBSD support, part 10  
 433508 FreeBSD support, part 11  
 433510 FreeBSD support, part 12  
 433801 PPC ISA 3.1 support is missing, part 10 (ISA 3.1 support complete)  
 433863 s390x: memcheck/tests/s390x/{cds,cs,csg} failures  
 434296 s390x: False-positive memcheck diagnostics from vector string instructions  
 434840 PPC64 darn instruction not supported  
 435665 PPC ISA 3.0 copy, paste, cpabort instructions are not supported  
 435908 valgrind tries to fetch from deubginfo for files which already have debug information  
 438871 unhandled instruction bytes: 0xF3 0x49 0xF 0x6F 0x9C 0x24 0x60 0x2  
 439046 valgrind is unusably large when linked with lld  
 439090 Implement close\_range(2)  
 439326 Valgrind 3.17.0 won't compile with Intel 2021 oneAPI compilers  
 439590 glibc-2.34 breaks suppressions against obj:\*/lib\*/libc-2.\*so\*  
 440670 unhandled ppc64le-linux syscall: 252 statfs64 and 253 fstatfs64  
 440906 Fix impossible constraint issue in P10 testcase.  
 441512 Remove a unneeded / unnecessary prefix check.  
 441534 Update the expected output for test\_isa\_3\_1\_VRT.  
 442061 very slow execution under Fedora 34 (readdwarf3)  
 443031 Gcc -many change requires explicit .machine directives  
 443033 Add support for the ISA 3.0 mcrxr instruction  
 443034 Sraw, srawi, sradi, sradi, mfs  
 443178 Powerpc, test jm-mfspr expected output needs to be updated.  
 443179 Need new test for the lxvx and stxvx instructions on ISA 2.07 and ISA 3.0 systems.  
 443180 The subnormal test and the ISA 3.0 test generate compiler warnings  
 443314 In the latest GIT version, Valgrind with "--trace-flags" crashes at "al" register

443605 Don't call final\_tidyup (\_\_libc\_freeres) on FatalSignal

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)

where XXXXXX is the bug number as listed below.

(3.18.0.RC1: 12 Oct 2021)

(3.18.0: 15 Oct 2021)

Release 3.17.0 (19 Mar 2021)

~~~~~

3.17.0 fixes a number of bugs and adds some functional changes: support for GCC 11, Clang 11, DWARF5 debuginfo, the 'debuginfod' debuginfo server, and some new instructions for Arm64, S390 and POWER. There are also some tool updates.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

\* ===== CORE CHANGES =====

\* DWARF version 5 support. Valgrind can now read DWARF version 5 debuginfo as produced by GCC 11.

\* Valgrind now supports debuginfod, an HTTP server for distributing ELF/DWARF debugging information. When a debuginfo file cannot be found locally, Valgrind is able to query debuginfod servers for the file using its build-id. See the user manual for more information about debuginfod support.

\* ===== PLATFORM CHANGES =====

\* arm64:

- Inaccuracies resulting from double-rounding in the simulation of floating-point multiply-add/subtract instructions have been fixed. These should now behave exactly as the hardware does.

- Partial support for the ARM v8.2 instruction set. v8.2 support work is ongoing. Support for the half-word variants of at least the following instructions has been added:

- FABS <Hd>, <Hn>
- FABS <Vd>.<T>, <Vn>.<T>
- FNEG <Hd>, <Hn>
- FNEG <Vd>.<T>, <Vn>.<T>
- FSQRT <Hd>, <Hn>
- FSQRT <Vd>.<T>, <Vn>.<T>
- FADDP

\* s390:

- Implement the new instructions/features that were added to z/Architecture with the vector-enhancements facility 1. Also cover the instructions from

the vector-packed-decimal facility that are defined outside the chapter "Vector Decimal Instructions", but not the ones from that chapter itself.

For a detailed list of newly supported instructions see the updates to ``docs/internals/s390-opcodes.csv'`.

Since the miscellaneous instruction extensions facility 2 was already added in Valgrind 3.16.0, this completes the support necessary to run general programs built with `--march=z14'` under Valgrind. The vector-packed-decimal facility is currently not exploited by the standard toolchain and libraries.

\* ppc64:

- Various bug fixes. Fix for the sync field to limit setting just two of the two bits in the L-field. Fix the write size for the stxsibx and stxsihx instructions. Fix the modsw and modsd instructions.
- Partial support for ISA 3.1 has been added. Support for the VSX PCV mask instructions, bfloat16 GER instructions, and bfloat16 to/from float 32-bit conversion instructions are still missing.

\* ===== TOOL CHANGES =====

\* General tool changes

- All the tools and their vgpreload libraries are now installed under libexec because they cannot be executed directly and should be run through the valgrind executable. This should be an internal, not user visible, change, but might impact valgrind packagers.
- The `--track-fds` option now respects `-q`, `--quiet` and won't output anything if no file descriptors are leaked. It also won't report the standard stdin (0), stdout (1) or stderr (2) descriptors as being leaked with `--trace-fds=yes` anymore. To track whether the standard file descriptors are still open at the end of the program run use `--trace-fds=all`.

\* DHAT:

- DHAT has been extended, with two new modes of operation. The new `--mode=copy` flag triggers copy profiling, which records calls to `memcpy`, `strcpy`, and similar functions. The new `--mode=ad-hoc` flag triggers ad hoc profiling, which records calls to the `DHAT_AD_HOC_EVENT` client request in the new `dhat/dhat.h` file. This is useful for learning more about hot code paths. See the user manual for more information about the new modes.
- Because of these changes, DHAT's file format has changed. DHAT output files produced with earlier versions of DHAT will not work with this version of DHAT's viewer, and DHAT output files produced with this version of DHAT will not work with earlier versions of DHAT's viewer.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that

are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)

where XXXXXX is the bug number as listed below.

140178 `open("/proc/self/exe", ...)`; doesn't quite work  
 140939 `--track-fds` reports leakage of `stdout/in/err` and doesn't respect `-q`  
 217695 `malloc/calloc/realloc/memalign` failure doesn't set `errno` to `ENOMEM`  
 338633 `gdbserver_tests/nlcontrolc.vgtest` hangs on arm64  
 345077 linux syscall `execveat` support (linux 3.19)  
 361770 Missing `F_ADD_SEALS`  
 369029 handle linux syscalls `sched_getattr` and `sched_setattr`  
 384729 `__libc_freeres` inhibits cross-platform `valgrind`  
 388787 Support for C++17 `new/delete`  
 391853 `Makefile.all.am:L247` and `@SOLARIS_UNDEF_LARGESOURCE@` being empty  
 396656 Warnings while reading debug info  
 397605 `ioctl FICLONE` mishandled  
 401416 Compile failure with `openmpi 4.0`  
 408663 Suppression file for `musl libc`  
 404076 s390x: z14 vector instructions not implemented  
 410743 `shmat()` calls for 32-bit programs fail when running in 64-bit `valgrind`  
     (actually affected all x86 and nanomips regardless of host bitness)  
 413547 regression test does not check for Arm 64 features.  
 414268 Enable `AArch64` feature detection and decoding for `v8.x` instructions  
 415293 Incorrect call-graph tracking due to new `_dl_runtime_resolve_xsave*`  
 422174 unhandled instruction bytes: `0x48 0xE9` (REX prefixed `JMP` instruction)  
 422261 platform selection fails for unqualified client name  
 422623 `epoll_ctl` warns for uninitialized padding on non-amd64 64bit arches  
 423021 PPC: Add missing ISA 3.0 documentation link and `HWCAPS` test.  
 423195 PPC ISA 3.1 support is missing, part 1  
 423361 Adds `io_uring` support on arm64/aarch64 (and all other arches)  
 424012 crash with `readv/writev` having invalid but not `NULL` `arg2` `iovec`  
 424298 amd64: Implement `RDSEED`  
 425232 PPC ISA 3.1 support is missing, part 2  
 425820 Failure to recognize `vpcmpeqq` as a dependency breaking idiom.  
 426014 arm64: implement `fmadd` and `fmsub` as `Iop_MAdd/Sub`  
 426123 PPC ISA 3.1 support is missing, part 3  
 426144 Fix "condition variable has not been initialized" on Fedora 33.  
 427400 PPC ISA 3.1 support is missing, part 4  
 427401 PPC ISA 3.1 support is missing, part 5  
 427404 PPC ISA 3.1 support is missing, part 6  
 427870 `lmw`, `lswi` and related PowerPC insns aren't allowed on `ppc64le`  
 427787 Support new `faccessat2` linux syscall (439)  
 427969 `debuginfo` section duplicates a section in the main ELF file  
 428035 drd: Unbreak the `musl` build  
 428648 `s390_emit_load_mem` panics due to 20-bit offset for vector load  
 428716 `cppcheck` detects potential leak in `VEX/useful/smchash.c`  
 428909 `helgrind`: need to intercept duplicate `libc` definitions for Fedora 33  
 429352 PPC ISA 3.1 support is missing, part 7  
 429354 PPC ISA 3.1 support is missing, part 8  
 429692 unhandled `ppc64le-linux` syscall: 147 (`getsid`)  
 429864 s390x: C++ atomic `test_and_set` yields false-positive `memcheck`  
     diagnostics  
 429952 Errors when building `regtest` with `clang`  
 430354 `ppc stxsibx` and `stxsihx` instructions write too much data  
 430429 `valgrind.h` doesn't compile on s390x with `clang`  
 430485 `expr_is_guardable` doesn't handle `Iex_Qop`

431556 Complete arm64 FADDP v8.2 instruction support  
 432102 Add support for DWARF5 as produced by GCC11  
 432161 Addition of arm64 v8.2 FADDP, FNEG and FSQRT  
 432381 drd: Process STACK\_REGISTER client requests  
 432552 [AArch64] invalid error emitted for pre-decremented byte/hword addresses  
 432672 vg\_regtest: test-specific environment variables not reset between tests  
 432809 VEX should support REX.W + POPF  
 432861 PPC modsw and modsd give incorrect results for 1 mod 12  
 432870 gdbserver\_tests:nlcontrolc hangs with newest glibc2.33 x86-64  
 432215 Add debuginfod functionality  
 433323 Use pkglibexecdir as vglbdir  
 433500 DRD regtest faulures when libstdc++ and libgcc debuginfo are installed  
 433629 valgrind/README has type "abd" instead of "and"  
 433641 Rust std::sys::unix::fs::try\_statx Syscall param fstatat(file\_name)  
 433898 arm64: Handle sp, lr, fp as DwReg in CfiExpr  
 434193 GCC 9+ inlined strcmp causes "Conditional jump or move [...] value" report  
 n-i-bz helgrind: If hg\_cli\_\_realloc fails, return NULL.  
 n-i-bz arm64 front end: avoid Memcheck false positives relating to CPUID

(3.17.0.RC1: 13 Mar 2021)  
 (3.17.0.RC2: 17 Mar 2021)  
 (3.17.0: 19 Mar 2021)

#### Release 3.16.1 (22 June 2020)

~~~~~

3.16.1 fixes two critical bugs discovered after 3.16.0 was frozen. It also fixes character encoding problems in the documentation HTML.

422677 PPC sync instruction L field should only be 2 bits in ISA 3.0  
 422715 32-bit x86: vex: the 'impossible' happened: expr\_is\_guardable: unhandled expr

(3.16.1, 22 June 2020, 36d6727e1d768333a536f274491e5879cab2c2f7)

#### Release 3.16.0 (27 May 2020)

~~~~~

3.16.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13 and nanoMIPS/Linux.

\* ===== CORE CHANGES =====

\* It is now possible to dynamically change the value of many command line options while your program (or its children) are running under Valgrind.

To see the list of dynamically changeable options, run  
 "valgrind --help-dyn-options".

You can change the options from the shell by using `vgdb` to launch the monitor command `"v.clo <clo option>..."`.

The same monitor command can be used from a `gdb` connected to the `valgrind gdbserver`.

Your program can also change the dynamically changeable options using the client request `VALGRIND_CLO_CHANGE(option)`.

\* ===== PLATFORM CHANGES =====

\* MIPS: preliminary support for nanoMIPS instruction set has been added.

\* ===== TOOL CHANGES =====

\* DHAT:

- The implicit memcpy done by each call to `realloc` now counts towards the read and write counts of resized heap blocks, making those counts higher and more accurate.

\* Cachelgrind:

- `cg_annotate's` `--auto` and `--show-percs` options now default to 'yes', because they are usually wanted.

\* Callgrind:

- `callgrind_annotate's` `--auto` and `--show-percs` options now default to 'yes', because they are usually wanted.
- The command option `--collect-systime` has been enhanced to specify the unit used to record the elapsed time spent during system calls. The command option now accepts the values `no|yes|msec|usec|nsec`, where `yes` is a synonym of `msec`. When giving the value `nsec`, the system cpu time of system calls is also recorded.

\* Memcheck:

- Several memcheck options are now dynamically changeable. Use `valgrind --help-dyn-options` to list them.
- The release 3.15 introduced a backward incompatible change for some suppression entries related to `preadv` and `pwritev` syscalls. When reading a suppression entry using the unsupported 3.14 format, `valgrind` will now produce a warning to say the suppression entry will not work, and suggest the needed change.
- Significantly fewer false positive errors on optimised code generated by Clang and GCC. In particular, Memcheck now deals better with the situation where the compiler will transform C-level `"A && B"` into `"B && A"` under certain circumstances (in which the transformation is valid). Handling of integer equality/non-equality checks on partially defined values is also improved on some architectures.

\* exp-sgcheck:

- The experimental Stack and Global Array Checking tool has been removed. It only ever worked on x86 and amd64, and even on those it had a high false positive rate and was slow. An alternative for detecting

stack and global array overruns is using the AddressSanitizer (ASAN) facility of the GCC and Clang compilers, which require you to rebuild your code with `-fsanitize=address`.

\* ===== OTHER CHANGES =====

\* New and modified GDB server monitor features:

- Option `-T` tells `vgdb` to output a timestamp in the `vgdb` information messages.
- The `gdbserver` monitor commands that require an address and an optional length argument now accepts the alternate 'C like' syntax `"address[length]"`. For example, the `memcheck` command `"monitor who_points_at 0x12345678 120"` can now also be given as `"monitor who_points_at 0x12345678[120]"`.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)

where XXXXXX is the bug number as listed below.

343099 Linux `setns` syscall wrapper missing, unhandled syscall: 308  
 == 368923 WARNING: unhandled arm64-linux syscall: 268 (setns)  
 == 369031 WARNING: unhandled amd64-linux syscall: 308 (setns)  
 385386 Assertion failed "`szB >= CACHE_ENTRY_SIZE`" at `m_debuginfo/image.c:517`  
 400162 Patch: Guard against `__GLIBC_PREREQ` for musl libc  
 400593 In Coregrind, use `statx` for some internal syscalls if `[f]stat[64]` fail  
 400872 Add nanoMIPS support to Valgrind  
 403212 `drd/tests/trylock` hangs on FreeBSD  
 404406 s390x: z14 miscellaneous instructions not implemented  
 405201 Incorrect size of struct `vki_siginfo` on 64-bit Linux architectures  
 406561 `mcinfcallsr` `gdbserver_test` fails on ppc64  
 406824 Unsupported baseline  
 407218 Add support for the `copy_file_range` syscall  
 407307 Intercept `stpcpy` also in `ld.so` for arm64  
 407376 Update Xen support to 4.12 (4.13, actually) and add more coverage  
 == 390553  
 407764 `drd cond_post_wait` gets wrong (?) condition on s390x z13 system  
 408009 Expose `rdrand` and `f16c` even on `avx` if host cpu supports them  
 408091 Missing `pkey` syscalls  
 408414 Add support for missing `preadv2` and `pwritev2` syscalls  
 409141 Valgrind hangs when SIGKILLed  
 409206 Support for Linux PPS and PTP ioctls  
 409367 `exit_group()` after signal to thread waiting in `futex()` causes hangs  
 409429 amd64: recognize 'cmpeq' variants as a dependency breaking idiom  
 409780 References to non-existent `configure.in`  
 410556 Add support for `BLKIO{MIN,OPT}` and `BLKALIGNOFF` ioctls  
 410599 Non-deterministic behaviour of `pth_self_kill_15_other` test  
 410757 discrepancy for `preadv2/pwritev2` syscalls across different versions  
 411134 Allow the user to change a set of command line options during execution  
 411451 amd64->IR of `bt/btc/bts/btr` with immediate clears zero flag

412344 Problem setting mips flags with specific paths  
 412408 unhandled arm-linux syscall: 124 - adjtime - on arm-linux  
 413119 Ioctl wrapper for DRM\_IOCTL\_I915\_GEM\_MMAP  
 413330 avx-1 test fails on AMD EPYC 7401P 24-Core Processor  
 413603 callgrind\_annotate/cg\_annotate truncate function names at '#'  
 414565 Specific use case bug found in SysRes VG\_(do\_sys\_sigprocmask)  
 415136 ARMv8.1 Compare-and-Swap instructions are not supported  
 415757 vex x86->IR: 0x66 0xF 0xCE 0x4F (bswapw)  
 416239 valgrind crashes when handling clock\_adjtime  
 416285 Use prlimit64 in VG\_(getrlimit) and VG\_(setrlimit)  
 416286 DRD reports "conflicting load" error on std::mutex::lock()  
 416301 s390x: "compare and signal" not supported  
 416387 finit\_module and bpf syscalls are unhandled on arm64  
 416464 Fix false reports for uninitialized memory for PR\_CAPBSET\_READ/DROP  
 416667 gcc10 ppc64le impossible constraint in 'asm' in test\_isa.  
 416753 new 32bit time syscalls for 2038+  
 417075 pwritev(vector[...]) suppression ignored  
     417075 is not fixed, but incompatible supp entries are detected  
     and a warning is produced for these.  
 417187 [MIPS] Conditional branch problem since 'grail' changes  
 417238 Test memcheck/tests/vbit-test fails on mips64 BE  
 417266 Make memcheck/tests/linux/sigqueue usable with musl  
 417281 s390x: /bin/true segfaults with "grail" enabled  
 417427 commit to fix vki\_siginfo\_t definition created numerous regression  
     errors on ppc64  
 417452 s390\_insn\_store\_emit: dst->tag for HRcVec128  
 417578 Add suppressions for glibc DTV leaks  
 417906 clone with CLONE\_VFORK and no CLONE\_VM fails  
 418004 Grail code additions break ppc64.  
 418435 s390x: spurious "Conditional jump or move depends on uninitialised [..]"  
 418997 s390x: Support lex\_ITE for float and vector types  
 419503 s390x: Avoid modifying registers returned from isel functions  
 421321 gcc10 arm64 build needs \_\_getauxval for linking with libgcc  
 421570 std\_mutex fails on Arm v8.1 h/w  
 434035 vgdb might crash if valgrind is killed  
 n-i-bz Fix minor one time leaks in dhat.  
 n-i-bz Add --run-cxx-freeres=no in outer args to avoid inner crashes.  
 n-i-bz Add support for the Linux io\_uring system calls  
 n-i-bz sys\_statx: don't complain if both |filename| and |buf| are NULL.  
 n-i-bz Fix non-glibc build of test suite with s390x\_features  
 n-i-bz MinGW, include/valgrind.h: Fix detection of 64-bit mode  
 423195 PPC ISA 3.1 support is missing, part 1

(3.16.0.RC1: 18 May 2020, git 6052ee66a0cf5234e8e2a2b49a8760226bc13b92)  
 (3.16.0.RC2: 19 May 2020, git 940ec1ca69a09f7fdae3e800b7359f85c13c4b37)  
 (3.16.0: 27 May 2020, git bf5e647edb9e96cbd5c57cc944984402eeee296d)

## Release 3.15.0 (12 April 2019)

~~~~~

3.15.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android,

X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12. There is also preliminary support for X86/macOS 10.13 and AMD64/macOS 10.13.

\* ===== CORE CHANGES =====

\* The XTree Massif output format now makes use of the information obtained when specifying `--read-inline-info=yes`.

\* amd64 (x86\_64): the RDRAND and F16C insn set extensions are now supported.

\* ===== TOOL CHANGES =====

\* DHAT:

- DHAT been thoroughly overhauled, improved, and given a GUI. As a result, it has been promoted from an experimental tool to a regular tool. Run it with `--tool=dhat` instead of `--tool=exp-dhat`.
- DHAT now prints only minimal data when the program ends, instead writing the bulk of the profiling data to a file. As a result, the `--show-top-n` and `--sort-by` options have been removed.
- Profile results can be viewed with the new viewer, `dh_view.html`. When a run ends, a short message is printed, explaining how to view the result.
- See the documentation for more details.

\* Cachegrind:

- `cg_annotate` has a new option, `--show-percs`, which prints percentages next to all event counts.

\* Callgrind:

- `callgrind_annotate` has a new option, `--show-percs`, which prints percentages next to all event counts.
- `callgrind_annotate` now inserts commas in call counts, and sort the caller/callee lists in the call tree.

\* Massif:

- The default value for `--read-inline-info` is now "yes" on Linux/Android/Solaris. It is still "no" on other OS.

\* Memcheck:

- The option `--xtree-leak=yes` (to output leak result in xtree format) automatically activates the option `--show-leak-kinds=all`, as xtree visualisation tools such as `kcachegrind` can in any case select what kind of leak to visualise.
- There has been further work to avoid false positives. In particular, integer equality on partially defined inputs (`C ==` and `!=`) is now handled better.

\* ===== OTHER CHANGES =====

\* The new option `--show-error-list=no|yes` displays, at the end of the run, the list of detected errors and the used suppressions. Prior to this change, showing this information could only be done by specifying `"-v -v"`, but that also produced a lot of other possibly-non-useful messages. The option `-s` is equivalent to `--show-error-list=yes`.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
where XXXXXX is the bug number as listed below.

385411 s390x: z13 vector floating-point instructions not implemented  
397187 z13 vector register support for vgdb gdbserver  
398183 Vex errors with `_mm256_shuffle_epi8/vpshufb`  
398870 Please add support for instruction `vcvtps2ph`  
399287 amd64 front end: Illegal Instruction `vcmpttrueps`  
399301 Use inlined frames in Massif XTree output.  
399322 Improve `callgrind_annotate` output  
399444 VEX/priv/guest\_s390\_toIR.c:17407: (style) Mismatching assignment [...]  
400164 helgrind test encounters mips x-compiler warnings and assembler error  
400490 s390x: VRs allocated as if separate from FPRs  
400491 s390x: Operand of LOCH treated as unsigned integer  
400975 Compile error: error: '-mips64r2' conflicts with the other architecture options, which specify a mips64 processor  
401112 LLVM 5.0 generates comparison against partially initialized data  
401277 More bugs in z13 support  
401454 Add a `--show-percs` option to `cg_annotate` and `callgrind_annotate`.  
401578 drd: crashes sometimes on `fork()`  
401627 memcheck errors with glibc `avx2` optimized `wcsncmp`  
401822 none/tests/ppc64/jm-vmx fails and produces assembler warnings  
401827 none/tests/ppc64/test\_isa\_2\_06\_part3 failure on ppc64le (xvrsqrtesp)  
401828 none/tests/ppc64/test\_isa\_2\_06\_part1 failure on ppc64le (fcfidus and fcfidus)  
402006 mark helper regs defined in `final_tidyup` before `freeres_wrapper` call  
402048 WARNING: unhandled ppc64[be|le]-linux syscall: 26 (ptrace)  
402123 invalid assembler opcodes for mips32r2  
402134 assertion fail in `mc_translate.c` (noteTmpUsesIn) `Iex_VECRET` on arm64  
402327 Warning: DWARF2 CFI reader: unhandled DW\_OP\_ opcode 0x13 (DW\_OP\_drop)  
402341 drd/tests/tsan\_thread\_wrappers\_pthread.h:369: suspicious code ?  
402351 mips64 libvexmultiarch\_test fails on s390x  
402369 Overhaul DHAT  
402395 coregrind/vgdb-invoker-solaris.c: 2 \* poor error checking  
402480 Do not use `%rsp` in clobber list  
402481 vbit-test fails on x86 for `Iop_CmpEQ64 iselInt64Expr Sar64`  
402515 Implement new option `--show-error-list=no|yes / -s`  
402519 POWER 3.0 addex instruction incorrectly implemented  
402781 Redo the cache used to process indirect branch targets  
403123 vex amd64->IR:0xF3 0x48 0xF 0xAE 0xD3 (wrfibase)  
403552 s390x: wrong facility bit checked for vector facility  
404054 memcheck powerpc subfe x, x, x initializes x to 0 or -1 based on CA

404638 Add VG\_(replaceIndexXA)  
 404843 s390x: backtrace sometimes ends prematurely  
 404888 autotools cleanup series  
 405079 unhandled ppc64le-linux syscall: 131 (quotactl)  
 405182 Valgrind fails to build with Clang  
 405205 filter\_libc: remove the line holding the futex syscall error entirely  
 405356 PPC64, xvcvsxdsp, xvcvxdsp are supposed to write the 32-bit result to the upper and lower 32-bits of the 64-bit result  
 405362 PPC64, vmsummbm instruction doesn't handle overflow case correctly  
 405363 PPC64, xvcvdpdxws, xvcvdpuxws, do not handle NaN arguments correctly.  
 405365 PPC64, function \_get\_maxmin\_fp\_NaN() doesn't handle QNaN, SNaN case correctly.  
 405403 s390x disassembler cannot be used on x86  
 405430 Use gcc -Wimplicit-fallthrough=2 by default if available  
 405458 MIPS mkFormVEC arguments swapped?  
 405716 drd: Fix an integer overflow in the stack margin calculation  
 405722 Support arm64 core dump  
 405733 PPC64, xvcvdpdp should write 32-bit result to upper and lower 32-bits of the 64-bit destination field.  
 405734 PPC64, vrlwnm, vrlwmi, vrldrm, vrlldmi do not work properly when me < mb  
 405782 "VEX temporary storage exhausted" when attempting to debug slic3r-pe  
 406198 none/tests/ppc64/test\_isa\_3\_0\_other test sporadically including CA bit in output.  
 406256 PPC64, vector floating point instructions don't handle subnormal according to VSCR[NJ] bit setting.  
 406352 cachegrind/callgrind fails ann tests because of missing a.c  
 406354 dhat is broken on x86 (32bit)  
 406355 mcsignopass, mcsigpass, mcbreak fail due to difference in gdb output  
 406357 gdbserver\_tests fails because of gdb output change  
 406360 memcheck/tests/libstdc++.supp needs more supression variants  
 406422 none/tests/amd64-linux/map\_32bits.vgtest fails too easily  
 406465 arm64 insn selector fails on "t0 = <expr>" where <expr> has type Ity\_F16  
 407340 PPC64, does not support the vlogef, vextefp instructions.  
 n-i-bz add syswrap for PTRACE\_GET|SET\_THREAD\_AREA on amd64.  
 n-i-bz Fix callgrind\_annotate non deterministic order for equal total  
 n-i-bz callgrind\_annotate --threshold=100 does not print all functions.  
 n-i-bz callgrind\_annotate Use of uninitialized value in numeric gt (>)  
 n-i-bz amd64 (x86\_64): RDRAND and F16C insn set extensions are supported

(3.15.0.RC1: 8 April 2019, git ce94d674de5b99df173aad4c3ee48fc2a92e5d9c)  
 (3.15.0.RC2: 11 April 2019, git 0c8be9bbbede189ec580ec270521811766429595f)  
 (3.15.0: 14 April 2019, git 270037da8b508954f0f7d703a0bebf5364eec548)

# Release 3.14.0 (9 October 2018)

~~~~~

3.14.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12. There is also preliminary support for X86/macOS 10.13, AMD64/macOS 10.13.

\* ===== CORE CHANGES =====

\* The new option `--keep-debuginfo=no|yes` (default no) can be used to retain debug info for unloaded code. This allows saved stack traces (e.g. for memory leaks) to include file/line info for code that has been dlclosed (or similar). See the user manual for more information and known limitations.

\* Ability to specify suppressions based on source file name and line number.

\* Majorly overhauled register allocator. No end-user changes, but the JIT generates code a bit more quickly now.

\* ===== PLATFORM CHANGES =====

\* Preliminary support for macOS 10.13 has been added.

\* mips: support for MIPS32/MIPS64 Revision 6 has been added.

\* mips: support for MIPS SIMD architecture (MSA) has been added.

\* mips: support for MIPS N32 ABI has been added.

\* s390: partial support for vector instructions (integer and string) has been added.

\* ===== TOOL CHANGES =====

\* Helgrind: Addition of a flag `--delta-stacktrace=no|yes` [yes on linux amd64/x86] which specifies how full history stack traces should be computed. Setting this to `=yes` can speed up Helgrind by 25% when using `--history-level=full`.

\* Memcheck: reduced false positive rate for optimised code created by Clang 6 / LLVM 6 on x86, amd64 and arm64. In particular, Memcheck analyses code blocks more carefully to determine where it can avoid expensive definedness checks without loss of precision. This is controlled by the flag `--expensive-definedness-checks=no|auto|yes` [auto].

\* ===== OTHER CHANGES =====

\* Valgrind is now buildable with link-time optimisation (LTO). A new configure option `--enable-lto=yes` allows building Valgrind with LTO. If the toolchain supports it, this produces a smaller/faster Valgrind (up to 10%). Note that if you are doing Valgrind development, `--enable-lto=yes` massively slows down the build process.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit [https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX) where XXXXXX is the bug number as listed below.

79362 Debug info is lost for .so files when they are dlclosed  
208052 strcpy error when n = 0  
255603 exp-sgcheck Assertion '!already\_present' failed  
338252 building valgrind with -flto (link time optimisation) fails  
345763 MIPS N32 ABI support  
368913 WARNING: unhandled arm64-linux syscall: 117 (ptrace)  
== 388664 unhandled arm64-linux syscall: 117 (ptrace)  
372347 Replacement problem of the additional c++14/c++17 new/delete operators  
373069 memcheck/tests/leak\_cpp\_interior fails with GCC 5.1+  
376257 helgrind history full speed up using a cached stack  
379373 Fix syscall param msg->desc.port.name points to uninitialised byte(s)  
on macOS 10.12  
379748 Fix missing pselect syscall (OS X 10.11)  
379754 Fix missing syscall uclock\_wait (OS X 10.12)  
380397 s390x: \_\_GI\_strerror() replacement needed  
381162 possible array overrun in VEX register allocator  
381272 ppc64 doesn't compile test\_isa\_2\_06\_partx.c without VSX support  
381274 powerpc too chatty even with --sigill-diagnostics=no  
381289 epoll\_pwait can have a NULL sigmask  
381553 VEX register allocator v3  
381556 arm64: Handle feature registers access on 4.11 Linux kernel or later  
381769 Use ucontext\_t instead of struct ucontext  
381805 arm32 needs ld.so index hardware for new glibc security fixes  
382256 gz compiler flag test doesn't work for gold  
382407 vg\_perf needs "--terse" command line option  
382515 "Assertion 'di->have\_dinfo' failed." on wine's dlls/mscoree/tests/[...]  
382563 MIPS MSA ASE support  
382998 xml-socket doesn't work  
383275 massif: m\_xarray.c:162 (ensureSpaceXA): Assertion '!xa->arr' failed  
383723 Fix missing kevent\_qos syscall (macOS 10.11)  
== 385604 illegal hardware instruction (OpenCV cv::namedWindow)  
384096 Mention AddrCheck at Memcheck's command line option [...]  
384230 vex x86->IR: 0x67 0xE8 0xAB 0x68  
== 384156 vex x86->IR: 0x67 0xE8 0x6B 0x6A  
== 386115 vex x86->IR: 0x67 0xE8 0xD3 0x8B any program  
== 388407 vex x86->IR: 0x67 0xE8 0xAB 0x29  
== 394903 vex x86->IR: 0x67 0xE8 0x1B 0xDA  
384337 performance improvements to VEX register allocator v2 and v3  
384526 reduce number of spill insns generated by VEX register allocator v3  
384584 Callee saved regs listed first for AMD64, X86, and PPC architectures  
384631 Sanitise client args as printed with -v  
384633 Add a simple progress-reporting facility  
384987 VEX regalloc: allocate caller-save registers for short lived vregs  
385055 PPC VEX temporary storage exhausted  
385182 PPC64 is missing support for the DSCR  
385183 PPC64, Add support for xscmpeqdp, xscmptdp, xscmpgedp, xsmincdp  
385207 PPC64, generate\_store\_FPRF() generates too many lops  
385208 PPC64, xxperm instruction exhausts temporary memory  
385210 PPC64, vpermr instruction could exhaust temporary memory  
385279 unhandled syscall: mach:43 (mach\_generate\_activity\_id)  
== 395136 valgrind: m\_syswrap/syswrap-main.c:438 (Bool\_eq\_Syscall[...])  
== 387045 Valgrind crashing on High Sierra when testing any newly [...]  
385334 PPC64, fix vpermr, xxperm, xxpermr mask value.  
385408 s390x: z13 vector "support" instructions not implemented  
385409 s390x: z13 vector integer instructions not implemented  
385410 s390x: z13 vector string instructions not implemented  
385412 s390x: new non-vector z13 instructions not implemented

385868 glibc ld.so \_dl\_runtime\_resolve\_avx\_slow conditional jump warning.  
 385912 none/tests/rlimit\_nofile fails on newer glibc/kernel.  
 385939 Optionally exit on the first error  
 386318 valgrind.org/info/tools.html is missing SGCheck  
 386425 running valgrind + wine on armv7l gives illegal opcode  
 386397 PPC64, valgrind truncates powerpc timebase to 32-bits.  
 387410 MIPSr6 support  
 387664 Memcheck: make expensive-definedness-checks be the default  
 387712 s390x cgijl reports Conditional jump depends on uninitialised value  
 387766 asm shifts cause false positive "Conditional jump or move depends on uninitialised value"  
 387773 .gnu\_debugaltlink paths resolve relative to .debug file, not symlink  
 388174 valgrind with Wine quits with "Assertion 'cfsi\_fits' failed"  
 388786 Support bpf syscall in amd64 Linux  
 388862 Add replacements for wmemchr and wcsnlen on Linux  
 389065 valgrind meets gcc flag -Wlogical-op  
 389373 exp-sgcheck the 'impossible' happened as Ist\_LoadG is not instrumented  
 390471 suppression by specification of source-file line number  
 390723 make xtree dump files world wide readable, similar to log files  
 391164 constraint bug in tests/ppc64/test\_isa\_2\_07\_part1.c for mtfprwa  
 391861 Massif Assertion 'n\_ips >= 1 && n\_ips <= VG\_(clo\_backtrace\_size)'  
 392118 unhandled amd64-linux syscall: 332 (statx)  
 392449 callgrind not clearing the number of calls properly  
 393017 Add missing support for xsmaxcdp instruction, bug fixes for xsmincdp, lxssp, stxssp and stxvl instructions.  
 393023 callgrind\_control risks using the wrong vgdb  
 393062 build-id ELF phdrs read causes "debuginfo reader: ensure\_valid failed"  
 393099 posix\_memalign() invalid write if alignment == 0  
 393146 failing assert "is\_DebugInfo\_active(di)"  
 395709 PPC64 is missing support for the xvnegsp instruction  
 395682 Accept read-only PT\_LOAD segments and .rodata by ld -z separate-code == 384727  
 396475 valgrind OS-X build: config.h not found (out-of-tree macOS builds)  
 395991 arm-linux: wine's unit tests enter a signal delivery loop [..]  
 396839 s390x: Trap instructions not implemented  
 396887 arch\_prctl should return EINVAL on unknown option  
 == 397286 crash before launching binary (Unsupported arch\_prctl option)  
 == 397393 valgrind: the 'impossible' happened: (Archlinux)  
 == 397521 valgrind: the 'impossible' happened: Unsupported [..]  
 396906 compile tests failure on mips32-linux: broken inline asm in tests on mips32-linux  
 397012 glibc ld.so uses arch\_prctl on i386  
 397089 amd64: Incorrect decoding of three-register vmovss/vmovsd opcode 11h  
 397354 utimensat should ignore timespec tv\_sec if tv\_nsec is UTIME\_NOW/OMIT  
 397424 glibc 2.27 and gdb\_server tests  
 398028 Assertion `cfsi\_fits` failing in simple C program  
 398066 s390x: cgijl dep1, 0 reports false unitialised values warning

n-i-bz Fix missing workq\_ops operations (macOS)  
 n-i-bz fix bug in strspn replacement  
 n-i-bz Add support for the Linux BLKFLSBUF ioctl  
 n-i-bz Add support for the Linux BLKREPORTZONE and BLKRESETZONE ioctls  
 n-i-bz Fix possible stack trashing by semctl syscall wrapping  
 n-i-bz Add support for the Linux membarrier() system call  
 n-i-bz x86 front end: recognise and handle UD2 correctly  
 n-i-bz Signal delivery for x86-linux: ensure that the stack pointer is correctly aligned before entering the handler.

(3.14.0.RC1: 30 September 2018, git c2aeea2d28acb0639bcc8cc1e4ab115067db1eae)  
 (3.14.0.RC2: 3 October 2018, git 3e214c4858a6fdd5697e767543a0c19e30505582)  
 (3.14.0: 9 October 2018, git 353a3587bb0e2757411f9138f5e936728ed6cc4f)

## Release 3.13.0 (15 June 2017)

~~~~~

3.13.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris and AMD64/MacOSX 10.12.

### \* ===== CORE CHANGES =====

- \* The translation cache size has been increased to keep up with the demands of large applications. The maximum number of sectors has increased from 24 to 48. The default number of sectors has increased from 16 to 32 on all targets except Android, where the increase is from 6 to 12.
- \* The amount of memory that Valgrind can use has been increased from 64GB to 128GB. In particular this means your application can allocate up to about 60GB when running on Memcheck.
- \* Valgrind's default load address has been changed from 0x3800'0000 to 0x5800'0000, so as to make it possible to load larger executables. This should make it possible to load executables of size at least 1200MB.
- \* A massive spaceleak caused by reading compressed debuginfo files has been fixed. Valgrind should now be entirely usable with gcc-7.0 "-gz" created debuginfo.
- \* The C++ demangler has been updated.
- \* Support for demangling Rust symbols has been added.

- \* A new representation of stack traces, the "XTree", has been added. An XTree is a tree of stacktraces with data associated with the stacktraces. This is used by various tools (Memcheck, Helgrind, Massif) to report on the heap consumption of your program. Reporting is controlled by the new options --xtree-memory=none|allocs|full and --xtree-memory-file=<file>.

A report can also be produced on demand using the gdbserver monitor command 'xtmemory [<filename>]>'. The XTree can be output in 2 formats: 'callgrind format' and 'massif format'. The existing visualisers for these formats (e.g. callgrind\_annotate, KCachegrind, ms\_print) can be used to visualise and analyse these reports.

Memcheck can also produce XTree leak reports using the Callgrind file format. For more details, see the user manual.

### \* ===== PLATFORM CHANGES =====

- \* ppc64: support for ISA 3.0B and various fixes for existing 3.0 support

- \* amd64: fixes for JIT failure problems on long AVX2 code blocks
- \* amd64 and x86: support for CET prefixes has been added
- \* arm32: a few missing ARMv8 instructions have been implemented
- \* arm64, mips64, mips32: an alternative implementation of Load-Linked and Store-Conditional instructions has been added. This is to deal with processor implementations that implement the LL/SC specifications strictly and as a result cause Valgrind to hang in certain situations. The alternative implementation is automatically enabled at startup, as required. You can use the option `--sim-hints=fallback-llsc` to force-enable it if you want.
- \* Support for OSX 10.12 has been improved.
- \* On Linux, clone handling has been improved to honour `CLONE_VFORK` that involves a child stack. Note however that `CLONE_VFORK | CLONE_VM` is handled like `CLONE_VFORK` (by removing `CLONE_VM`), so applications that depend on `CLONE_VM` exact semantics will (still) not work.
- \* The TileGX/Linux port has been removed because it appears to be both unused and unsupported.
- \* ===== TOOL CHANGES =====
- \* Memcheck:
  - Memcheck should give fewer false positives when running optimised Clang/LLVM generated code.
  - Support for `--xtree-memory` and `'xtmemory [<filename>]>'`.
  - New command line options `--xtree-leak=no|yes` and `--xtree-leak-file=<file>` to produce the end of execution leak report in a xtree callgrind format file.
  - New option `'xtleak'` in the memcheck `leak_check` monitor command, to produce the leak report in an xtree file.
- \* Massif:
  - Support for `--xtree-memory` and `'xtmemory [<filename>]>'`.
  - For some workloads (typically, for big applications), Massif memory consumption and CPU consumption has decreased significantly.
- \* Helgrind:
  - Support for `--xtree-memory` and `'xtmemory [<filename>]>'`.
  - addition of client request `VALGRIND_HG_GNAT_DEPENDENT_MASTER_JOIN`, useful for Ada gnat compiled applications.
- \* ===== OTHER CHANGES =====
- \* For Valgrind developers: in an outer/inner setup, the outer Valgrind will

append the inner guest stacktrace to the inner host stacktrace. This helps to investigate the errors reported by the outer, when they are caused by the inner guest program (such as an inner regtest). See README\_DEVELOPERS for more info.

\* To allow fast detection of callgrind files by desktop environments and file managers, the format was extended to have an optional first line that uniquely identifies the format ("# callgrind format"). Callgrind creates this line now, as does the new xtree functionality.

\* File name template arguments (such as --log-file, --xtree-memory-file, ...) have a new %n format letter that is replaced by a sequence number.

\* "--version -v" now shows the SVN revision numbers from which Valgrind was built.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)

where XXXXXX is the bug number as listed below.

162848 --log-file output isn't split when a program forks  
 340777 Illegal instruction on mips (ar71xx)  
 341481 MIPS64: Iop\_CmpNE32 triggers false warning on MIPS64 platforms  
 342040 Valgrind mishandles clone with CLONE\_VFORK | CLONE\_VM that clones to a different stack.  
 344139 x86 stack-seg overrides, needed by the Wine people  
 344524 store conditional of guest applications always fail - observed on Octeon3(MIPS)  
 348616 Wine/valgrind: noted but unhandled ioctl 0x5390 [...] (DVD\_READ\_STRUCT)  
 352395 Please provide SVN revision info in --version -v  
 352767 Wine/valgrind: noted but unhandled ioctl 0x5307 [...] (CDROMSTOP)  
 356374 Assertion 'DRD\_(g\_threadinfo)[tid].pt\_threadid != INVALID\_POSIX\_THREADID' failed  
 358213 helgrind/drd bar\_bad testcase hangs or crashes with new glibc pthread barrier implementation  
 358697 valgrind.h: Some code remains even when defining NVALGRIND  
 359202 Add musl libc configure/compile  
 360415 amd64 instructions ADCX and ADOX are not implemented in VEX  
 == 372828 (vex amd64->IR: 0x66 0xF 0x3A 0x62 0x4A 0x10)  
 360429 unhandled ioctl 0x530d with no size/direction hints (CDROMREADMODE1)  
 362223 assertion failed when .valgrindrc is a directory instead of a file  
 367543 bt/btc/btr/bts x86/x86\_64 instructions are poorly-handled wrt flags  
 367942 Segfault vgPlain\_do\_sys\_sigaction (m\_signals.c:1138)  
 368507 can't malloc chunks larger than about 34GB  
 368529 Android arm target link error, missing atexit and pthread\_atfork  
 368863 WARNING: unhandled arm64-linux syscall: 100 (get\_robust\_list)  
 368865 WARNING: unhandled arm64-linux syscall: 272 (kcmp)  
 368868 disInstr(arm64): unhandled instruction 0xD53BE000 = cntfrq\_el0 (ARMv8)  
 368917 WARNING: unhandled arm64-linux syscall: 218 (request\_key)

368918 WARNING: unhandled arm64-linux syscall: 127 (sched\_rr\_get\_interval)  
 368922 WARNING: unhandled arm64-linux syscall: 161 (sethostname)  
 368924 WARNING: unhandled arm64-linux syscall: 84 (sync\_file\_range)  
 368925 WARNING: unhandled arm64-linux syscall: 130 (tkill)  
 368926 WARNING: unhandled arm64-linux syscall: 97 (unshare)  
 369459 valgrind on arm64 violates the ARMv8 spec (ldxr/stxr)  
 370028 Reduce the number of compiler warnings on MIPS platforms  
 370635 arm64 missing syscall getcpu  
 371225 Fix order of timer\_{gettime,getoverrun,settime} syscalls on arm64  
 371227 Clean AArch64 syscall table  
 371412 Rename wrap\_sys\_shmat to sys\_shmat like other wrappers  
 371471 Valgrind complains about non legit memory leaks on placement new (C++)  
 371491 handleAddrOverrides() is [incorrect] when ASO prefix is used  
 371503 disInstr(arm64): unhandled instruction 0xF89F0000  
 371869 support '%' in symbol Z-encoding  
 371916 execution tree xtree concept  
 372120 c++ demangler demangles symbols which are not c++  
 372185 Support of valgrind on ARMv8 with 32 bit executable  
 372188 vex amd64->IR: 0x66 0xF 0x3A 0x62 0x4A 0x10 0x10 0x48 (PCMPxSTRx \$0x10)  
 372195 Power PC, xssel instruction is not always recognized.  
 372504 Hanging on exit\_group  
 372600 process loops forever when fatal signals are arriving quickly  
 372794 LibVEX (arm32 front end): 'Assertion szBlg2 <= 3' failed  
 373046 Stacks registered by core are never deregistered  
 373069 memcheck/tests/leak\_cpp\_interior fails with GCC 5.1+  
 373086 Implement additional Xen hypercalls  
 373192 Calling posix\_spawn in glibc 2.24 completely broken  
 373488 Support for fanotify API on ARM64 architecture  
 == 368864 WARNING: unhandled arm64-linux syscall: 262 (fanotify\_init)  
 373555 Rename BBPTR to GSPTR as it denotes guest state pointer only  
 373938 const IRExpr arguments for matchIRExpr()  
 374719 some spelling fixes  
 374963 increase valgrind's load address to prevent mmap failure  
 375514 valgrind\_get\_tls\_addr() does not work in case of static TLS  
 375772 +1 error in get\_elf\_symbol\_info() when computing value of 'hi' address  
 for ML\_(find\_rx\_mapping())  
 375806 Test helgrind/tests/tc22\_exit\_w\_lock fails with glibc 2.24  
 375839 Temporary storage exhausted, with long sequence of vfmadd231ps insns  
 == 377159 "vex: the 'impossible' happened" still present  
 == 375150 Assertion 'tres.status == VexTransOK' failed  
 == 378068 valgrind crashes on AVX2 function in FFMpeg  
 376142 Segfaults on MIPS Cavium Octeon boards  
 376279 disInstr(arm64): unhandled instruction 0xD50320FF  
 376455 Solaris: unhandled syscall lgrpsys(180)  
 376518 Solaris: unhandled fast trap getlgrp(6)  
 376611 ppc64 and arm64 don't know about prlimit64 syscall  
 376729 PPC64, remove R2 from the clobber list  
 == 371668  
 376956 syswrap of SNDDRV and DRM\_IOCTL\_VERSION causing some addresses  
 to be wrongly marked as addressable  
 377066 Some Valgrind unit tests fail to compile on Ubuntu 16.10 with  
 PIE enabled by default  
 377376 memcheck/tests/linux/getregset fails with glibc2.24  
 377427 PPC64, lxx instruction failing on odd destination register  
 377478 PPC64: ISA 3.0 setup fixes  
 377698 Missing memory check for futex() uaddr arg for FUTEX\_WAKE  
 and FUTEX\_WAKE\_BITSET, check only 4 args for FUTEX\_WAKE\_BITSET,  
 and 2 args for FUTEX\_TRYLOCK\_PI

377717 Fix massive space leak when reading compressed debuginfo sections  
 377891 Update Xen 4.6 domctl wrappers  
 377930 fcntl syscall wrapper is missing flock structure check  
 378524 libvexmultiarch\_test regression on s390x and ppc64  
 378535 Valgrind reports INTERNAL ERROR in execve syscall wrapper  
 378673 Update libiberty demangler  
 378931 Add ISA 3.0B additional instructions, add OV32, CA32 setting support  
 379039 syscall wrapper for prctl(PR\_SET\_NAME) must not check more than 16 bytes  
 379094 Valgrind reports INTERNAL ERROR in rt\_sigsuspend syscall wrapper  
 379371 UNKNOWN task message [id 3444, to mach\_task\_self(), reply 0x603]  
     (task\_register\_dyld\_image\_infos)  
 379372 UNKNOWN task message [id 3447, to mach\_task\_self(), reply 0x603]  
     (task\_register\_dyld\_shared\_cache\_image\_info)  
 379390 unhandled syscall: mach:70 (host\_create\_mach\_voucher\_trap)  
 379473 MIPS: add support for rdhwr cycle counter register  
 379504 remove TileGX/Linux port  
 379525 Support more x86 nop opcodes  
 379838 disAMode(x86): not an addr!  
 379703 PC ISA 3.0 fixes: stxvx, stxv, xscmpexpdp instructions  
 379890 arm: unhandled instruction: 0xEBAD 0x1B05 (sub.w fp, sp, r5, lsl #4)  
 379895 clock\_gettime does not execute POST syscall wrapper  
 379925 PPC64, mtffs does not set the FPCC and C bits in the FPSCR correctly  
 379966 WARNING: unhandled amd64-linux syscall: 313 (finit\_module)  
 380200 xtree generated callgrind files refer to files without directory name  
 380202 Assertion failure for cache line size (cls == 64) on aarch64.  
 380397 s390x: \_\_GI\_strerror() replacement needed  
 n-i-bz Fix pub\_tool\_basics.h build issue with g++ 4.4.7.

(3.13.0.RC1: 2 June 2017, vex r3386, valgrind r16434)  
 (3.13.0.RC2: 9 June 2017, vex r3389, valgrind r16443)  
 (3.13.0: 14 June 2017, vex r3396, valgrind r16446)

## Release 3.12.0 (20 October 2016)

~~~~~

3.12.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, X86/MacOSX 10.10 and AMD64/MacOSX 10.10. There is also preliminary support for X86/MacOSX 10.11/12, AMD64/MacOSX 10.11/12 and TILEGX/Linux.

## \* ===== PLATFORM CHANGES =====

- \* POWER: Support for ISA 3.0 has been added
- \* mips: support for O32 FPXX ABI has been added.
- \* mips: improved recognition of different processors
- \* mips: determination of page size now done at run time
- \* amd64: Partial support for AMD FMA4 instructions.
- \* arm, arm64: Support for v8 crypto and CRC instructions.

\* Improvements and robustification of the Solaris port.

\* Preliminary support for MacOS 10.12 (Sierra) has been added.

Whilst 3.12.0 continues to support the 32-bit x86 instruction set, we would prefer users to migrate to 64-bit x86 (a.k.a amd64 or x86\_64) where possible. Valgrind's support for 32-bit x86 has stagnated in recent years and has fallen far behind that for 64-bit x86 instructions. By contrast 64-bit x86 is well supported, up to and including AVX2.

\* ===== TOOL CHANGES =====

\* Memcheck:

- Added meta mempool support for describing a custom allocator which:
  - Auto-frees all chunks assuming that destroying a pool destroys all objects in the pool
  - Uses itself to allocate other memory blocks
- New flag `--ignore-range-below-sp` to ignore memory accesses below the stack pointer, if you really have to. The related flag `--workaround-gcc296-bugs=yes` is now deprecated. Use `--ignore-range-below-sp=1024-1` as a replacement.

\* DRD:

- Improved thread startup time significantly on non-Linux platforms.

\* DHAT

- Added collection of the metric "tot-blocks-allocd"

\* ===== OTHER CHANGES =====

\* Replacement/wrapping of malloc/new related functions is now done not just for system libraries by default, but for any globally defined malloc/new related function (both in shared libraries and statically linked alternative malloc implementations). The dynamic (runtime) linker is excluded, though. To only intercept malloc/new related functions in system libraries use `--soname-synonyms=somalloc=nouserintercepts` (where "nouserintercepts" can be any non-existing library name). This new functionality is not implemented for MacOS X.

\* The maximum number of callers in a suppression entry is now equal to the maximum size for `--num-callers` (500). Note that `--gen-suppressions=yes|all` similarly generates suppressions containing up to `--num-callers` frames.

\* New and modified GDB server monitor features:

- Valgrind's gdbserver now accepts the command 'catch syscall'. Note that you must have GDB `>= 7.11` to use 'catch syscall' with gdbserver.

\* New option `--run-cxx-freeres=<yes|no>` can be used to change whether `__gnu_cxx::__freeres()` cleanup function is called or not. Default is

'yes'.

- \* Valgrind is able to read compressed debuginfo sections in two formats:
  - zlib ELF gABI format with SHF\_COMPRESSED flag (gcc option -gz=zlib)
  - zlib GNU format with .zdebug sections (gcc option -gz=zlib-gnu)
- \* Modest JIT-cost improvements: the cost of instrumenting code blocks for the most common use case (x86\_64-linux, Memcheck) has been reduced by 10%-15%.
- \* Improved performance for programs that do a lot of discarding of instruction address ranges of 8KB or less.
- \* The C++ symbol demangler has been updated.
- \* More robustness against invalid syscall parameters on Linux.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit  
[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed below.

191069 Exiting due to signal not reported in XML output  
 199468 Suppressions: stack size limited to 25  
     while --num-callers allows more frames  
 212352 vex amd64 unhandled opc\_aux = 0x 2, first\_opcode == 0xDC (FCOM)  
 278744 cvtps2pd with redundant RexW  
 303877 valgrind doesn't support compressed debuginfo sections.  
 345307 Warning about "still reachable" memory when using libstdc++ from gcc 5  
 348345 Assertion fails for negative lineno  
 348924 MIPS: Load doubles through memory so the code compiles with the FPXX ABI  
 351282 V 3.10.1 MIPS softfloat build broken with GCC 4.9.3 / binutils 2.25.1  
 351692 Dumps created by valgrind are not readable by gdb (mips32 specific)  
 351804 Crash on generating suppressions for "printf" call on OS X 10.10  
 352197 mips: mmap2() not wrapped correctly for page size > 4096  
 353083 arm64 doesn't implement various xattr system calls  
 353084 arm64 doesn't support sigpending system call  
 353137 www: update info for Supported Platforms  
 353138 www: update "The Valgrind Developers" page  
 353370 don't advertise RDRAND in cpuid for Core-i7-4910-like avx2 machine  
     == 365325  
     == 357873  
 353384 amd64->IR: 0x66 0xF 0x3A 0x62 0xD1 0x62 (pcmpXstrX \$0x62)  
 353398 WARNING: unhandled amd64-solaris syscall: 207  
 353660 XML in auxwhat tag not escaping reserved symbols properly  
 353680 s390x: Crash with certain glibc versions due to non-implemented TBEGIN  
 353727 amd64->IR: 0x66 0xF 0x3A 0x62 0xD1 0x72 (pcmpXstrX \$0x72)  
 353802 ELF debug info reader confused with multiple .rodata sections  
 353891 Assert 'bad\_scanned\_addr < VG\_ROUNDNDN(start+len, sizeof(Addr))' failed  
 353917 unhandled amd64-solaris syscall fchdir(120)

353920 unhandled amd64-solaris syscall: 170  
 354274 arm: unhandled instruction: 0xEBAD 0x0AC1 (sub.w sl, sp, r1, lsl #3)  
 354392 unhandled amd64-solaris syscall: 171  
 354797 Vbit test does not include Iops for Power 8 instruction support  
 354883 tst->os\_state.pthread - magic\_delta assertion failure on OSX 10.11  
     == 361351  
     == 362920  
     == 366222  
 354933 Fix documentation of --kernel-variant=android-no-hw-tls option  
 355188 valgrind should intercept all malloc related global functions  
 355454 do not intercept malloc related symbols from the runtime linker  
 355455 stderr.exp of test cases wrapmalloc and wrapmallocstatic overconstrained  
 356044 Dwarf line info reader misinterprets is\_stmt register  
 356112 mips: replace addi with addiu  
 356393 valgrind (vex) crashes because isZeroU happened  
     == 363497  
     == 364497  
 356676 arm64-linux: unhandled syscalls 125, 126 (sched\_get\_priority\_max/min)  
 356678 arm64-linux: unhandled syscall 232 (mincore)  
 356817 valgrind.h triggers compiler errors on MSVC when defining NVALGRIND  
 356823 Unsupported ARM instruction: stlex  
 357059 x86/amd64: SSE cvtpi2ps with memory source does transition to MMX state  
 357338 Unhandled instruction for SHA instructions libcrypto Boring SSL  
 357673 crash if I try to run valgrind with a binary link with libcurl  
 357833 Setting RLIMIT\_DATA to zero breaks with linux 4.5+  
 357871 pthread\_spin\_destroy not properly wrapped  
 357887 Calls to VG\_(fclose) do not close the file descriptor  
 357932 amd64->IR: accept redundant REX prefixes for {minsd,maxsd} m128, xmm.  
 358030 support direct socket calls on x86 32bit (new in linux 4.3)  
 358478 drd/tests/std\_thread.cpp doesn't build with GCC6  
 359133 Assertion 'eltSzB <= ddpa->poolSzB' failed  
 359181 Buffer Overflow during Demangling  
 359201 futex syscall "skips" argument 5 if op is FUTEX\_WAIT\_BITSET  
 359289 s390x: popcnt (B9E1) not implemented  
 359472 The Power PC vsbuqmq instruction doesn't always give the correct result  
 359503 Add missing syscalls for aarch64 (arm64)  
 359645 "You need libc6-dbg" help message could be more helpful  
 359703 s390: wire up separate socketcalls system calls  
 359724 getsockname might crash - deref\_UInt should call safe\_to\_deref  
 359733 amd64 implement ld.so strchr/index override like x86  
 359767 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 1/5  
 359829 Power PC test suite none/tests/ppc64/test\_isa\_2\_07.c uses uninitialized data  
 359838 arm64: Unhandled instruction 0xD5033F5F (clrex)  
 359871 Incorrect mask handling in ppoll  
 359952 Unrecognised PCMPESTRM variants (0x70, 0x19)  
 360008 Contents of Power vr registers contents is not printed correctly when the --vgdb-shadow-registers=yes option is used  
 360035 POWER PC instruction bcdadd and bcdsubtract generate result with non-zero shadow bits  
 360378 arm64: Unhandled instruction 0x5E280844 (sha1h s4, s2)  
 360425 arm64 unsupported instruction ldpsw  
     == 364435  
 360519 none/tests/arm64/memory.vgtest might fail with newer gcc  
 360571 Error about the Android Runtime reading below the stack pointer on ARM  
 360574 Wrong parameter type for an ashmem ioctl() call on Android and ARM64  
 360749 kludge for multiple .rodata sections on Solaris no longer needed  
 360752 raise the number of reserved fds in m\_main.c from 10 to 12

361207 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 2/5  
 361226 s390x: risbgn (EC59) not implemented  
 361253 [s390x] ex\_clone.c:42: undefined reference to `pthread\_create'  
 361354 ppc64[le]: wire up separate socketcalls system calls  
 361615 Inconsistent termination for multithreaded process terminated by signal  
 361926 Unhandled Solaris syscall: sysfs(84)  
 362009 V dumps core on unimplemented functionality before threads are created  
 362329 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 3/5  
 362894 missing (broken) support for wbit field on mtfspi instruction (ppc64)  
 362935 [AsusWRT] Assertion 'sizeof(TTEntryC) <= 88' failed  
 362953 Request for an update to the Valgrind Developers page  
 363680 add renameat2() support  
 363705 arm64 missing syscall name\_to\_handle\_at and open\_by\_handle\_at  
 363714 ppc64 missing syscalls sync, waitid and name\_to/open\_by\_handle\_at  
 363858 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 4/5  
 364058 clarify in manual limitations of array overruns detections  
 364413 pselect syscallwrapper mishandles NULL sigmask  
 364728 Power PC, missing support for several HW registers in  
     get\_otrack\_shadow\_offset\_wrk()  
 364948 Valgrind does not support the IBM POWER ISA 3.0 instructions, part 5/5  
 365273 Invalid write to stack location reported after signal handler runs  
 365912 ppc64BE segfault during jm-insns test (RELRO)  
 366079 FPXX Support for MIPS32 Valgrind  
 366138 Fix configure errors out when using Xcode 8 (clang 8.0.0)  
 366344 Multiple unhandled instruction for Aarch64  
     (0x0EE0E020, 0x1AC15800, 0x4E284801, 0x5E040023, 0x5E056060)  
 367995 Integration of memcheck with custom memory allocator  
 368120 x86\_linux asm \_start functions do not keep 16-byte aligned stack pointer  
 368412 False positive result for altivec capability check  
 368416 Add tc06\_two\_races\_xml.exp output for ppc64  
 368419 Perf Events ioctls not implemented  
 368461 mmapunmap test fails on ppc64  
 368823 run\_a\_thread\_NORETURN assembly code typo for VGP\_arm64\_linux target  
 369000 AMD64 fma4 instructions unsupported.  
 369169 ppc64 fails jm\_int\_isa\_2\_07 test  
 369175 jm\_vec\_isa\_2\_07 test crashes on ppc64  
 369209 valgrind loops and eats up all memory if cwd doesn't exist.  
 369356 pre\_mem\_read\_sockaddr syscall wrapper can crash with bad sockaddr  
 369359 msghdr\_foreachfield can crash when handling bad iovec  
 369360 Bad sigprocmask old or new sets can crash valgrind  
 369361 vmsplice syscall wrapper crashes on bad iovec  
 369362 Bad sigaction arguments crash valgrind  
 369383 x86 sys\_modify\_ldt wrapper crashes on bad ptr  
 369402 Bad set/get\_thread\_area pointer crashes valgrind  
 369441 bad lvec argument crashes process\_vm\_readv/writev syscall wrappers  
 369446 valgrind crashes on unknown fcntl command  
 369439 S390x: Unhandled insns RISBLG/RISBHG and LDE/LDER  
 369468 Remove quadratic metapool algorithm using VG\_(HT\_remove\_at\_iter)  
 370265 ISA 3.0 HW cap stuff needs updating  
 371128 BCD add and subtract instructions on Power BE in 32-bit mode do not work  
 372195 Power PC, xxsel instruction is not always recognized  
  
 n-i-bz Fix incorrect (or infinite loop) unwind on RHEL7 x86 and amd64  
 n-i-bz massif --pages-as-heap=yes does not report peak caused by mmap+munmap  
 n-i-bz false positive leaks due to aspacemgr merging heap & non heap segments  
 n-i-bz Fix ppoll\_alarm exclusion on OS X  
 n-i-bz Document brk segment limitation, reference manual in limit reached msg.  
 n-i-bz Fix clobber list in none/tests/amd64/xacq\_xrel.c [valgrind r15737]

n-i-bz Bump allowed shift value for "add.w reg, sp, reg, lsl #N" [vex r3206]  
 n-i-bz amd64: memcheck false positive with shr %edx  
 n-i-bz arm3: Allow early writeback of SP base register in "strd rD, [sp, #-16]"  
 n-i-bz ppc: Fix two cases of PPCAvFpOp vs PPCFpOp enum confusion  
 n-i-bz arm: Fix incorrect register-number constraint check for LDAEX{,B,H,D}  
 n-i-bz DHAT: added collection of the metric "tot-blocks-allocd"

(3.12.0.RC1: 20 October 2016, vex r3282, valgrind r16094)  
 (3.12.0.RC2: 20 October 2016, vex r3282, valgrind r16096)  
 (3.12.0: 21 October 2016, vex r3282, valgrind r16098)

## Release 3.11.0 (22 September 2015)

~~~~~

3.11.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, X86/MacOSX 10.10 and AMD64/MacOSX 10.10. There is also preliminary support for X86/MacOSX 10.11, AMD64/MacOSX 10.11 and TILEGX/Linux.

### \* ===== PLATFORM CHANGES =====

- \* Support for Solaris/x86 and Solaris/amd64 has been added.
- \* Preliminary support for Mac OS X 10.11 (El Capitan) has been added.
- \* Preliminary support for the Tilera TileGX architecture has been added.
- \* s390x: It is now required for the host to have the "long displacement" facility. The oldest supported machine model is z990.
- \* x86: on an SSE2 only host, Valgrind in 32 bit mode now claims to be a Pentium 4. 3.10.1 wrongly claimed to be a Core 2, which is SSSE3.
- \* The JIT's register allocator is significantly faster, making the JIT as a whole somewhat faster, so JIT-intensive activities, for example program startup, are modestly faster, around 5%.
- \* There have been changes to the default settings of several command line flags, as detailed below.
- \* Intel AVX2 support is more complete (64 bit targets only). On AVX2 capable hosts, the simulated CPUID will now indicate AVX2 support.

### \* ===== TOOL CHANGES =====

#### \* Memcheck:

- The default value for --leak-check-heuristics has been changed from "none" to "all". This helps to reduce the number of possibly lost blocks, in particular for C++ applications.

- The default value for `--keep-stacktraces` has been changed from "malloc-then-free" to "malloc-and-free". This has a small cost in memory (one word per malloc-ed block) but allows Memcheck to show the 3 stacktraces of a dangling reference: where the block was allocated, where it was freed, and where it is accessed after being freed.
- The default value for `--partial-loads-ok` has been changed from "no" to "yes", so as to avoid false positive errors resulting from some kinds of vectorised loops.
- A new monitor command 'xb <addr> <len>' shows the validity bits of <len> bytes at <addr>. The monitor command 'xb' is easier to use than `get_vbits` when you need to associate byte data value with their corresponding validity bits.
- The 'block\_list' monitor command has been enhanced:
  - o it can print a range of loss records
  - o it now accepts an optional argument 'limited <max\_blocks>' to control the number of blocks printed.
  - o if a block has been found using a heuristic, then 'block\_list' now shows the heuristic after the block size.
  - o the loss records/blocks to print can be limited to the blocks found via specified heuristics.
- The C helper functions used to instrument loads on x86-{linux,solaris} and arm-linux (both 32-bit only) have been replaced by handwritten assembly sequences. This gives speedups in the region of 0% to 7% for those targets only.
- A new command line option, `--expensive-definedness-checks=yes|no`, has been added. This is useful for avoiding occasional invalid uninitialised-value errors in optimised code. Watch out for runtime degradation, as this can be up to 25%. As always, though, the slowdown is highly application specific. The default setting is "no".

\* Massif:

- A new monitor command 'all\_snapshots <filename>' dumps all snapshots taken so far.

\* Helgrind:

- Significant memory reduction and moderate speedups for `--history-level=full` for applications accessing a lot of memory with many different stacktraces.
- The default value for `--conflict-cache-size=N` has been doubled to 2000000. Users that were not using the default value should preferably also double the value they give.

The default was changed due to the changes in the "full history" implementation. Doubling the value gives on average a slightly more complete history and uses similar memory (or significantly less memory in the worst case) than the previous implementation.

- The Helgrind monitor command 'info locks' now accepts an optional argument 'lock\_addr', which shows information about the lock at the

given address only.

- When using `--history-level=full`, the new Helgrind monitor command `'accesshistory <addr> [<len>]'` will show the recorded accesses for `<len>` (or 1) bytes at `<addr>`.

\* ===== OTHER CHANGES =====

- \* The default value for the `--smc-check` option has been changed from "stack" to "all-non-file" on targets that provide automatic D-I cache coherence (x86, amd64 and s390x). The result is to provide, by default, transparent support for JIT generated and self-modifying code on all targets.
- \* Mac OS X only: the default value for the `--dsymutil` option has been changed from "no" to "yes", since any serious usage on Mac OS X always required it to be "yes".
- \* The command line options `--db-attach` and `--db-command` have been removed. They were deprecated in 3.10.0.
- \* When a process dies due to a signal, Valgrind now shows the signal and the stacktrace at default verbosity (i.e. verbosity 1).
- \* The address description logic used by Memcheck and Helgrind now describes addresses in anonymous segments, file mmap-ed segments, shared memory segments and the brk data segment.
- \* The new option `--error-markers=<begin>,<end>` can be used to mark the begin/end of errors in textual output mode, to facilitate searching/extracting errors in output files that mix valgrind errors with program output.
- \* The new option `--max-threads=<number>` can be used to change the number of threads valgrind can handle. The default is 500 threads which should be more than enough for most applications.
- \* The new option `--valgrind-stacksize=<number>` can be used to change the size of the private thread stacks used by Valgrind. This is useful for reducing memory use or increasing the stack size if Valgrind segfaults due to stack overflow.
- \* The new option `--avg-transtab-entry-size=<number>` can be used to specify the expected instrumented block size, either to reduce memory use or to avoid excessive retranslation.
- \* Valgrind can be built with Intel's ICC compiler, version 14.0 or later.
- \* New and modified GDB server monitor features:
  - When a signal is reported in GDB, you can now use the GDB convenience variable `$_siginfo` to examine detailed signal information.
  - Valgrind's gdbserver now allows the user to change the signal to deliver to the process. So, use 'signal SIGNAL' to continue execution with SIGNAL instead of the signal reported to GDB. Use 'signal 0' to continue without passing the signal to the process.

- With GDB >= 7.10, the command 'target remote' will automatically load the executable file of the process running under Valgrind. This means you do not need to specify the executable file yourself, GDB will discover it itself. See GDB documentation about 'qXfer:exec-file:read' packet for more info.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit  
[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed below.

116002 VG\_(printf): Problems with justification of strings and integers  
 155125 avoid cutting away file:lineno after long function name  
 197259 Unsupported arch\_ptrctl PR\_SET\_GS option  
 201152 ppc64: Assertion in ppc32g\_dirtyhelper\_MFSPR\_268\_269  
 201216 Fix Valgrind does not support pthread\_sigmask() on OS X  
 201435 Fix Darwin: -v does not show kernel version  
 208217 "Warning: noted but unhandled ioctl 0x2000747b" on Mac OS X  
 211256 Fixed an outdated comment regarding the default platform.  
 211529 Incomplete call stacks for code compiled by newer versions of MSVC  
 211926 Avoid compilation warnings in valgrind.h with -pedantic  
 212291 Fix unhandled syscall: unix:132 (mkfifo) on OS X  
 == 263119  
 226609 Crediting upstream authors in man page  
 231257 Valgrind omits path when executing script from shebang line  
 254164 OS X task\_info: UNKNOWN task message [id 3405, to mach\_task\_self() [...]]  
 294065 Improve the pdb file reader by avoiding hardwired absolute pathnames  
 269360 s390x: Fix addressing mode selection for compare-and-swap  
 302630 Memcheck: Assertion failed: 'sizeof(UWord) == sizeof(UInt)'  
 == 326797  
 312989 ioctl handling needs to do POST handling on generic ioctls and [...]  
 319274 Fix unhandled syscall: unix:410 (sigsuspend\_nocancel) on OS X  
 324181 mmap does not handle MAP\_32BIT (handle it now, rather than fail it)  
 327745 Fix valgrind 3.9.0 build fails on Mac OS X 10.6.8  
 330147 libmpiwrap PMPI\_Get\_count returns undefined value  
 333051 mmap of huge pages fails due to incorrect alignment  
 == 339163  
 334802 valgrind does not always explain why a given option is bad  
 335618 mov.w rN, pc/sp (ARM32)  
 335785 amd64->IR 0xC4 0xE2 0x75 0x2F (vmaskmovpd)  
 == 307399  
 == 343175  
 == 342740  
 == 346912  
 335907 segfault when running wine's ddrawex/tests/surface.c under valgrind  
 338602 AVX2 bit in CPUID missing  
 338606 Strange message for scripts with invalid interpreter  
 338731 ppc: Fix testuite build for toolchains not supporting -maltivec  
 338995 shmat with hugepages (SHM\_HUGETLB) fails with EINVAL  
 339045 Getting valgrind to compile and run on OS X Yosemite (10.10)

```

== 340252
339156 gdbsrv not called for fatal signal
339215 Valgrind 3.10.0 contain 2013 in copyrights notice
339288 support Cavium Octeon MIPS specific BBIT*32 instructions
339636 Use fxsave64 and fxrstor64 mnemonics instead of old-school rex64 prefix
339442 Fix testsuite build failure on OS X 10.9
339542 Enable compilation with Intel's ICC compiler
339563 The DVB demux DMX_STOP ioctl doesn't have a wrapper
339688 Mac-specific ASM does not support .version directive (cpuid,
    tronical and pushfpopf tests)
339745 Valgrind crash when check Marmalade app (partial fix)
339755 Fix known deliberate memory leak in setenv() on Mac OS X 10.9
339778 Linux/TileGx platform support for Valgrind
339780 Fix known uninitialised read in pthread_rwlock_init() on Mac OS X 10.9
339789 Fix none/tests/execve test on Mac OS X 10.9
339808 Fix none/tests/rlimit64_nofile test on Mac OS X 10.9
339820 vex amd64->IR: 0x66 0xF 0x3A 0x63 0xA 0x42 0x74 0x9 (pcmpistri $0x42)
340115 Fix none/tests/cmdline[1|2] tests on systems which define TMPDIR
340392 Allow user to select more accurate definedness checking in memcheck
    to avoid invalid complaints on optimised code
340430 Fix some grammatical weirdness in the manual.
341238 Recognize GCC5/DWARFv5 DW_LANG constants (Go, C11, C++11, C++14)
341419 Signal handler ucontext_t not filled out correctly on OS X
341539 VG_(describe_addr) should not describe address as belonging to client
    segment if it is past the heap end
341613 Enable building of manythreads and thread-exits tests on Mac OS X
341615 Fix none/tests/darwin/access_extended test on Mac OS X
341698 Valgrind's AESKEYGENASSIST gives wrong result in words 0 and 2 [...]
341789 aarch64: shmat fails with valgrind on ARMv8
341997 MIPS64: Cavium OCTEON insns - immediate operand handled incorrectly
342008 valgrind.h needs type cast [...] for clang/llvm in 64-bit mode
342038 Unhandled syscalls on aarch64 (mbind/get/set_mempolicy)
342063 wrong format specifier for test mcblocklistsearch in gdbserver_tests
342117 Hang when loading PDB file for MSVC compiled Firefox under Wine
342221 socket connect false positive uninit memory for unknown af family
342353 Allow dumping full massif output while valgrind is still running
342571 Valgrind chokes on AVX compare intrinsic with _CMP_GE_QS
== 346476
== 348387
== 350593
342603 Add I2C_SMBUS ioctl support
342635 OS X 10.10 (Yosemite) - missing system calls and fcntl code
342683 Mark memory past the initial brk limit as unaddressable
342783 arm: unhandled instruction 0xEEFE1ACA = "vcvt.s32.f32 s3, s3, #12"
342795 Internal glibc __GI_mempcpy call should be intercepted
342841 s390x: Support instructions fiebr(a) and fidbr(a)
343012 Unhandled syscall 319 (memfd_create)
343069 Patch updating v4l2 API support
343173 helgrind crash during stack unwind
343219 fix GET_STARTREGS for arm
343303 Fix known deliberate memory leak in setenv() on Mac OS X 10.10
343306 OS X 10.10: UNKNOWN mach_msg unhandled MACH_SEND_TRAILER option
343332 Unhandled instruction 0x9E310021 (fcvtmu) on aarch64
343335 unhandled instruction 0x1E638400 (fccmp) aarch64
343523 OS X mach_ports_register: UNKNOWN task message [id 3403, to [...]]
343525 OS X host_get_special_port: UNKNOWN host message [id 412, to [...]]
343597 ppc64le: incorrect use of offseof macro
343649 OS X host_create_mach_voucher: UNKNOWN host message [id 222, to [...]]

```

343663 OS X 10.10 Memcheck always reports a leak regardless of [...]  
 343732 Unhandled syscall 144 (setgid) on aarch64  
 343733 Unhandled syscall 187 (msgctl and related) on aarch64  
 343802 s390x: False positive "conditional jump or move depends on [...]  
 343902 --vgdb=yes doesn't break when --xml=yes is used  
 343967 Don't warn about setuid/setgid/setcap executable for directories  
 343978 Recognize DWARF5/GCC5 DW\_LANG\_Fortran 2003 and 2008 constants  
 344007 accept4 syscall unhandled on arm64 (242) and ppc64 (344)  
 344033 Helgrind on ARM32 loses track of mutex state in pthread\_cond\_wait  
 344054 www - update info for Solaris/illumos  
 344416 'make regtest' does not work cleanly on OS X  
 344235 Remove duplicate include of pub\_core\_aspacemgr.h  
 344279 syscall sendmmsg on arm64 (269) and ppc32/64 (349) unhandled  
 344295 syscall recvmmsg on arm64 (243) and ppc32/64 (343) unhandled  
 344307 2 unhandled syscalls on aarch64/arm64: umount2(39), mount (40)  
 344314 callgrind\_annotate ... warnings about commands containing newlines  
 344318 socketcall should wrap recvmmsg and sendmmsg  
 344337 Fix unhandled syscall: mach:41 (\_kernelrpc\_mach\_port\_guard\_trap)  
 344416 Fix 'make regtest' does not work cleanly on OS X  
 344499 Fix compilation for Linux kernel >= 4.0.0  
 344512 OS X: unhandled syscall: unix:348 (\_\_pthread\_chdir),  
         unix:349 (\_\_pthread\_fchdir)  
 344559 Garbage collection of unused segment names in address space manager  
 344560 Fix stack traces missing penultimate frame on OS X  
 344621 Fix memcheck/tests/err\_disable4 test on OS X  
 344686 Fix suppression for pthread\_rwlock\_init on OS X 10.10  
 344702 Fix missing libobjc suppressions on OS X 10.10  
         == 344543  
 344936 Fix unhandled syscall: unix:473 (readlinkat) on OS X 10.10  
 344939 Fix memcheck/tests/xml1 on OS X 10.10  
 345016 helgrind/tests/locked\_vs\_unlocked2 is failing sometimes  
 345079 Fix build problems in VEX/useful/test\_main.c  
 345126 Incorrect handling of VIDIOC\_G\_AUDIO and G\_AUDOUT  
 345177 arm64: prfm (reg) not implemented  
 345215 Performance improvements for the register allocator  
 345248 add support for Solaris OS in valgrind  
 345338 TIOCGSERIAL and TIOCSSERIAL ioctl support on Linux  
 345394 Fix memcheck/tests/strchr on OS X  
 345637 Fix memcheck/tests/sendmsg on OS X  
 345695 Add POWERPC support for AT\_DCACHESIZE and HWCAP2  
 345824 Fix aspacem segment mismatch: seen with none/tests/bigcode  
 345887 Fix an assertion in the address space manager  
 345928 amd64: callstack only contains current function for small stacks  
 345984 disInstr(arm): unhandled instruction: 0xEE193F1E  
 345987 MIPS64: Implement cavium LHX instruction  
 346031 MIPS: Implement support for the CvmCount register (rhwr %0, 31)  
 346185 Fix typo saving altivec register v24  
 346267 Compiler warnings for PPC64 code on call to LibVEX\_GuestPPC64\_get\_XER()  
         and LibVEX\_GuestPPC64\_get\_CR()  
 346270 Regression tests none/tests/jm\_vec/isa\_2\_07 and  
         none/tests/test\_isa\_2\_07\_part2 have failures on PPC64 little endian  
 346307 fuse filesystem syscall deadlocks  
 346324 PPC64 missing support for lbarx, lharx, stbcx and sthcx instructions  
 346411 MIPS: SysRes::\_valEx handling is incorrect  
 346416 Add support for LL\_IOC\_PATH2FID and LL\_IOC\_GETPARENT Lustre ioctls  
 346474 PPC64 Power 8, spr TEXASRU register not supported  
 346487 Compiler generates "note" about a future ABI change for PPC64  
 346562 MIPS64: lwl/lwr instructions are performing 64bit loads

and causing spurious "invalid read of size 8" warnings

346801 Fix link error on OS X: \_vgModuleLocal\_sf\_maybe\_extend\_stack

347151 Fix suppression for pthread\_rwlock\_init on OS X 10.8

347233 Fix memcheck/tests/strchr on OS X 10.10 (Haswell)

347322 Power PC regression test cleanup

347379 valgrind --leak-check=full leak errors from system libs on OS X 10.8  
== 217236

347389 unhandled syscall: 373 (Linux ARM syncfs)

347686 Patch set to cleanup PPC64 regtests

347978 Remove bash dependencies where not needed

347982 OS X: undefined symbols for architecture x86\_64: "\_global" [...]

347988 Memcheck: the 'impossible' happened: unexpected size for Addr (OSX/wine)  
== 345929

348102 Patch updating v4l2 API support

348247 amd64 front end: jno jumps wrongly when overflow is not set

348269 Improve mmap MAP\_HUGETLB support.

348334 (ppc) valgrind does not simulate dcbfl - then my program terminates

348345 Assertion fails for negative lineno

348377 Unsupported ARM instruction: yield

348565 Fix detection of command line option availability for clang

348574 vex amd64->IR: pcmpistri SSE4.2 unsupported (pcmpistri \$0x18)

348728 Fix broken check for VIDIOC\_G\_ENC\_INDEX

348748 Fix redundant condition

348890 Fix clang warning about unsupported --param inline-unit-growth=900

348949 Bogus "ERROR: --ignore-ranges: suspiciously large range"

349034 Add Lustre ioctls LL\_IOC\_GROUP\_LOCK and LL\_IOC\_GROUP\_UNLOCK

349086 Fix UNKNOWN task message [id 3406, to mach\_task\_self(), [...]]

349087 Fix UNKNOWN task message [id 3410, to mach\_task\_self(), [...]]

349626 Implemented additional Xen hypercalls

349769 Clang/osx: ld: warning: -read\_only\_relocs cannot be used with x86\_64

349790 Clean up of the hardware capability checking utilities.

349828 memcpy intercepts memmove causing src/dst overlap error (ppc64 ld.so)

349874 Fix typos in source code

349879 memcheck: add handwritten assembly for helperc\_LOADV\*

349941 di\_notify\_mmap might create wrong start/size DebugInfoMapping

350062 vex x86->IR: 0x66 0xF 0x3A 0xB (ROUNDSD) on OS X

350202 Add limited param to 'monitor block\_list'

350290 s390x: Support instructions fixbr(a)

350359 memcheck/tests/x86/fixsave hangs indefinitely on OS X

350809 Fix none/tests/async-sigs for Solaris

350811 Remove reference to --db-attach which has been removed.

350813 Memcheck/x86: enable handwritten assembly helpers for x86/Solaris too

350854 hard-to-understand code in VG\_(load\_ELF)()

351140 arm64 syscalls setuid (146) and setresgid (149) not implemented

351386 Solaris: Cannot run ld.so.1 under Valgrind

351474 Fix VG\_(iseqsigset) as obvious

351531 Typo in /include/vki/vki-xen-physdev.h header guard

351756 Intercept platform\_memchr\$VARIANT\$Haswell on OS X

351858 ldsoexec support on Solaris

351873 Newer gcc doesn't allow \_\_builtin\_tabortdc[i] in ppc32 mode

352130 helgrind reports false races for printf's using mempcpy on FILE\* state

352284 s390: Conditional jump depends on uninitialised value(s) in vfprintf

352320 arm64 crash on none/tests/nestedfs

352765 Vbit test fails on Power 6

352768 The mbar instruction is missing from the Power PC support

352769 Power PC program priority register (PPR) is not supported

n-i-bz Provide implementations of certain compiler builtins to support  
compilers that may not provide those

n-i-bz Old STABS code is still being compiled, but never used. Remove it.  
 n-i-bz Fix compilation on distros with glibc < 2.5  
 n-i-bz (vex 3098) Avoid generation of Neon insns on non-Neon hosts  
 n-i-bz Enable rt\_sigpending syscall on ppc64 linux.  
 n-i-bz mmap did not work properly on shared memory  
 n-i-bz Fix incorrect sizeof expression in syswrap-xen.c reported by Coverity  
 n-i-bz In VALGRIND\_PRINTF write out thread name, if any, to xml

(3.11.0.TEST1: 8 September 2015, vex r3187, valgrind r15646)  
 (3.11.0.TEST2: 21 September 2015, vex r3193, valgrind r15667)  
 (3.11.0: 22 September 2015, vex r3195, valgrind r15674)

#### Release 3.10.1 (25 November 2014)

~~~~~  
 3.10.1 is a bug fix release. It fixes various bugs reported in 3.10.0 and backports fixes for all reported missing AArch64 ARMv8 instructions and syscalls from the trunk. If you package or deliver 3.10.0 for others to use, you might want to consider upgrading to 3.10.1 instead.

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit  
[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed below.

335440 arm64: ld1 (single structure) is not implemented  
 335713 arm64: unhandled instruction: prfm (immediate)  
 339020 ppc64: memcheck/tests/ppc64/power\_ISA2\_05 failing in nightly build  
 339182 ppc64: AvSplat ought to load destination vector register with [..]  
 339336 PPC64 store quad instruction (stq) is not supposed to change [..]  
 339433 ppc64 lxvw4x instruction uses four 32-byte loads  
 339645 Use correct tag names in sys\_getdents/64 wrappers  
 339706 Fix false positive for ioctl(TIOCSIG) on linux  
 339721 assertion 'check\_sibling == sibling' failed in readdwarf3.c ...  
 339853 arm64 times syscall unknown  
 339855 arm64 unhandled getsid/setuid syscalls  
 339858 arm64 dmb sy not implemented  
 339926 Unhandled instruction 0x1E674001 (frintx) on aarm64  
 339927 Unhandled instruction 0x9E7100C6 (fcvtmu) on aarch64  
 339938 disInstr(arm64): unhandled instruction 0x4F8010A4 (fmla)  
 == 339950  
 339940 arm64: unhandled syscall: 83 (sys\_fdatasync) + patch  
 340033 arm64: unhandled insn dmb ishld and some other isb-dmb-dsb variants  
 340028 unhandled syscalls for arm64 (msync, pread64, setreuid and setregid)  
 340036 arm64: Unhandled instruction ld4 (multiple structures, no offset)  
 340236 arm64: unhandled syscalls: mkodat, fchdir, chroot, fchownat  
 340509 arm64: unhandled instruction fcvtas  
 340630 arm64: fchmod (52) and fchown (55) syscalls not recognized  
 340632 arm64: unhandled instruction fcvtas  
 340722 Resolve "UNKNOWN attrlist flags 0:0x10000000"  
 340725 AVX2: Incorrect decoding of vpbroadcast{b,w} reg,reg forms

340788 warning: unhandled syscall: 318 (getrandom)  
 340807 disInstr(arm): unhandled instruction: 0xEE989B20  
 340856 disInstr(arm64): unhandled instruction 0x1E634C45 (fcsel)  
 340922 arm64: unhandled getgroups/setgroups syscalls  
 350251 Fix typo in VEX utility program (test\_main.c).  
 350407 arm64: unhandled instruction ucvtf (vector, integer)  
 350809 none/tests/async-sigs breaks when run under cron on Solaris  
 350811 update README.solaris after r15445  
 350813 Use handwritten memcheck assembly helpers on x86/Solaris [...]  
 350854 strange code in VG\_(load\_ELF)()  
 351140 arm64 syscalls setuid (146) and setresgid (149) not implemented  
 n-i-bz DRD and Helgrind: Handle Imbe\_CancelReservation (clrex on ARM)  
 n-i-bz Add missing ]] to terminate CDATA.  
 n-i-bz Glibc versions prior to 2.5 do not define PTRACE\_GETSIGINFO  
 n-i-bz Enable sys\_fadvise64\_64 on arm32.  
 n-i-bz Add test cases for all remaining AArch64 SIMD, FP and memory insns.  
 n-i-bz Add test cases for all known arm64 load/store instructions.  
 n-i-bz PRE(sys\_openat): when checking whether ARG1 == VKI\_AT\_FDCWD [...]  
 n-i-bz Add detection of old ppc32 magic instructions from bug 278808.  
 n-i-bz exp-dhat: Implement missing function "dh\_malloc\_usable\_size".  
 n-i-bz arm64: Implement "fcvtu w, s".  
 n-i-bz arm64: implement ADDP and various others  
 n-i-bz arm64: Implement {S,U}CVTF (scalar, fixedpt).  
 n-i-bz arm64: enable FCVT{A,N}S X,S.

(3.10.1: 25 November 2014, vex r3026, valgrind r14785)

#### Release 3.10.0 (10 September 2014)

~~~~~

3.10.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, MIPS32/Android, X86/Android, X86/MacOSX 10.9 and AMD64/MacOSX 10.9. Support for MacOSX 10.8 and 10.9 is significantly improved relative to the 3.9.0 release.

#### \* ===== PLATFORM CHANGES =====

\* Support for the 64-bit ARM Architecture (AArch64 ARMv8). This port is mostly complete, and is usable, but some SIMD instructions are as yet unsupported.

\* Support for little-endian variant of the 64-bit POWER architecture.

\* Support for Android on MIPS32.

\* Support for 64bit FPU on MIPS32 platforms.

\* Both 32- and 64-bit executables are supported on MacOSX 10.8 and 10.9.

\* Configuration for and running on Android targets has changed. See README.android in the source tree for details.

\* ===== DEPRECATED FEATURES =====

\* --db-attach is now deprecated and will be removed in the next valgrind feature release. The built-in GDB server capabilities are superior and should be used instead. Learn more here:  
<http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver>

\* ===== TOOL CHANGES =====

\* Memcheck:

- Client code can now selectively disable and re-enable reporting of invalid address errors in specific ranges using the new client requests VALGRIND\_DISABLE\_ADDR\_ERROR\_REPORTING\_IN\_RANGE and VALGRIND\_ENABLE\_ADDR\_ERROR\_REPORTING\_IN\_RANGE.
- Leak checker: there is a new leak check heuristic called "length64". This is used to detect interior pointers pointing 8 bytes inside a block, on the assumption that the first 8 bytes holds the value "block size - 8". This is used by sqlite3MemMalloc, for example.
- Checking of system call parameters: if a syscall parameter (e.g. bind struct sockaddr, sendmsg struct msghdr, ...) has several fields not initialised, an error is now reported for each field. Previously, an error was reported only for the first uninitialised field.
- Mismatched alloc/free checking: a new flag --show-mismatched-frees=no|yes [yes] makes it possible to turn off such checks if necessary.

\* Helgrind:

- Improvements to error messages:
  - o Race condition error message involving heap allocated blocks also show the thread number that allocated the raced-on block.
  - o All locks referenced by an error message are now announced. Previously, some error messages only showed the lock addresses.
  - o The message indicating where a lock was first observed now also describes the address/location of the lock.
- Helgrind now understands the Ada task termination rules and creates a happens-before relationship between a terminated task and its master. This avoids some false positives and avoids a big memory leak when a lot of Ada tasks are created and terminated. The interceptions are only activated with forthcoming releases of gnatpro >= 7.3.0w-20140611 and gcc >= 5.0.
- A new GDB server monitor command "info locks" giving the list of locks, their location, and their status.

\* Callgrind:

- callgrind\_control now supports the --vgdb-prefix argument,

which is needed if valgrind was started with this same argument.

\* ===== OTHER CHANGES =====

- \* Unwinding through inlined function calls. Stack unwinding can now make use of Dwarf3 inlined-unwind information if it is available. The practical effect is that inlined calls become visible in stack traces. The suppression matching machinery has been adjusted accordingly. This is controlled by the new option `--read-inline-info=yes|no`. Currently this is enabled by default only on Linux and Android targets and only for the tools Memcheck, Helgrind and DRD.
- \* Valgrind can now read EXIDX unwind information on 32-bit ARM targets. If an object contains both CFI and EXIDX unwind information, Valgrind will prefer the CFI over the EXIDX. This facilitates unwinding through system libraries on arm-android targets.
- \* Address description logic has been improved and is now common between Memcheck and Helgrind, resulting in better address descriptions for some kinds of error messages.
- \* Error messages about dubious arguments (eg, to malloc or calloc) are output like other errors. This means that they can be suppressed and they have a stack trace.
- \* The C++ demangler has been updated for better C++11 support.
- \* New and modified GDB server monitor features:
  - Thread local variables/storage (`__thread`) can now be displayed.
  - The GDB server monitor command `"v.info location <address>"` displays information about an address. The information produced depends on the tool and on the options given to valgrind. Possibly, the following are described: global variables, local (stack) variables, allocated or freed blocks, ...
  - The option `"--vgdb-stop-at=event1,event2,..."` allows the user to ask the GDB server to stop at the start of program execution, at the end of the program execution and on Valgrind internal errors.
  - A new monitor command `"v.info stats"` shows various Valgrind core and tool statistics.
  - A new monitor command `"v.set hostvisibility"` allows the GDB server to provide access to Valgrind internal host status/memory.
- \* A new option `"--aspace-minaddr=<address>"` can in some situations allow the use of more memory by decreasing the address above which Valgrind maps memory. It can also be used to solve address conflicts with system libraries by increasing the default value. See user manual for details.
- \* The amount of memory used by Valgrind to store debug info (unwind info, line number information and symbol data) has been significantly reduced, even though Valgrind now reads more

information in order to support unwinding of inlined function calls.

\* Dwarf3 handling with `--read-var-info=yes` has been improved:

- Ada and C struct containing VLAs no longer cause a "bad DIE" error
- Code compiled with `-ffunction-sections -fdata-sections -Wl,--gc-sections` no longer causes assertion failures.

\* Improved checking for the `--sim-hints=` and `--kernel-variant=` options. Unknown strings are now detected and reported to the user as a usage error.

\* The semantics of stack start/end boundaries in the `valgrind.h` `VALGRIND_STACK_REGISTER` client request has been clarified and documented. The convention is that start and end are respectively the lowest and highest addressable bytes of the stack.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
where XXXXXX is the bug number as listed below.

175819 Support for ipv6 socket reporting with `--track-fds`  
 232510 make distcheck fails  
 249435 Analyzing wine programs with callgrind triggers a crash  
 278972 support for inlined function calls in stacktraces and suppression  
     == 199144  
 291310 FXSAVE instruction marks memory as undefined on amd64  
 303536 ioctl for SIOCETHTOOL (ethtool(8)) isn't wrapped  
 308729 vex x86->IR: unhandled instruction bytes 0xf 0x5 (syscall)  
 315199 vgcore file for threaded app does not show which thread crashed  
 315952 tun/tap ioctls are not supported  
 323178 Unhandled instruction: PLDW register (ARM)  
 323179 Unhandled instruction: PLDW immediate (ARM)  
 324050 Helgrind: SEGV because of unaligned stack when using movdqa  
 325110 Add test-cases for Power ISA 2.06 insns: divdo/divdo. and divduo/divduo.  
 325124 [MIPSEL] Compilation error  
 325477 Phase 4 support for IBM Power ISA 2.07  
 325538 cavium octeon mips64, valgrind reported "dumping core" [...]  
 325628 Phase 5 support for IBM Power ISA 2.07  
 325714 Empty vgcore but RLIMIT\_CORE is big enough (too big)  
 325751 Missing the two privileged Power PC Transactional Memory Instructions  
 325816 Phase 6 support for IBM Power ISA 2.07  
 325856 Make SGCheck fail gracefully on unsupported platforms  
 326026 Iop names for count leading zeros/sign bits incorrectly imply [...]  
 326436 DRD: False positive in libstdc++ `std::list::push_back`  
 326444 Cavium MIPS Octeon Specific Load Indexed Instructions  
 326462 Refactor vgdb to isolate invoker stuff into separate module

326469 amd64->IR: 0x66 0xF 0x3A 0x63 0xC1 0xE (pcmpistri 0x0E)  
326623 DRD: false positive conflict report in a field assignment  
326724 Valgrind does not compile on OSX 1.9 Mavericks  
326816 Intercept for \_\_strncpy\_sse2\_unaligned missing?  
326921 coregrind fails to compile m\_trampoline.S with MIPS/Linux port of V  
326983 Clear direction flag after tests on amd64.  
327212 Do not prepend the current directory to absolute path names.  
327223 Support for Cavium MIPS Octeon Atomic and Count Instructions  
327238 Callgrind Assertion 'passed <= last\_bb->cjmp\_count' failed  
327284 s390x: Fix translation of the risbg instruction  
327639 vex amd64->IR pcmpestri SSE4.2 instruction is unsupported 0x34  
327837 dwz compressed alternate .debug\_info and .debug\_str not read correctly  
327916 DW\_TAG\_typedef may have no name  
327943 s390x: add a redirection for the 'index' function  
328100 XABORT not implemented  
328205 Implement additional Xen hypercalls  
328454 add support Backtraces with ARM unwind tables (EXIDX)  
328455 s390x: SIGILL after emitting wrong register pair for ldxb  
328711 valgrind.1 manpage "memcheck options" section is badly generated  
328878 vex amd64->IR pcmpestri SSE4.2 instruction is unsupported 0x14  
329612 Incorrect handling of AT\_BASE for image execution  
329694 clang warns about using uninitialized variable  
329956 valgrind crashes when lmw/stmw instructions are used on ppc64  
330228 mmap must align to VKI\_SHMLBA on mips32  
330257 LLVM does not support `-mno-dynamic-no-pic` option  
330319 amd64->IR: unhandled instruction bytes: 0xF 0x1 0xD5 (xend)  
330459 --track-fds=yes doesn't track eventfds  
330469 Add clock\_adjtime syscall support  
330594 Missing sysalls on PowerPC / uClibc  
330622 Add test to regression suite for POWER instruction: dcbzl  
330939 Support for AMD's syscall instruction on x86  
== 308729  
330941 Typo in PRE(poll) syscall wrapper  
331057 unhandled instruction: 0xEE01B20 (vfma.f64) (has patch)  
331254 Fix expected output for memcheck/tests/dw4  
331255 Fix race condition in test none/tests/cool\_sigaction  
331257 Fix type of jump buffer in test none/tests/faultstatus  
331305 configure uses bash specific syntax  
331337 s390x WARNING: unhandled syscall: 326 (dup3)  
331380 Syscall param timer\_create(ev) points to uninitialised byte(s)  
331476 Patch to handle ioctl 0x5422 on Linux (x86 and amd64)  
331829 Unexpected ioctl opcode sign extension  
331830 ppc64: WARNING: unhandled syscall: 96/97  
331839 drd/tests/sem\_open specifies invalid semaphore name  
331847 outcome of drd/tests/thread\_name is nondeterministic  
332037 Valgrind cannot handle Thumb "add pc, reg"  
332055 drd asserts on platforms with VG\_STACK\_REDZONE\_SZB == 0 and  
consistency checks enabled  
332263 intercepts for pthread\_rwlock\_timedrdlock and  
pthread\_rwlock\_timedwrlock are incorrect  
332265 drd could do with post-rwlock\_init and pre-rwlock\_destroy  
client requests  
332276 Implement additional Xen hypercalls  
332658 ldrd.w r1, r2, [PC, #imm] does not adjust for 32bit alignment  
332765 Fix ms\_print to create temporary files in a proper directory  
333072 drd: Add semaphore annotations  
333145 Tests for misaligned PC+#imm access for arm  
333228 AAarch64 Missing instruction encoding: mrs %[reg], ctr\_el0

333230 AAarch64 missing instruction encodings: dc, ic, dsb.  
333248 WARNING: unhandled syscall: unix:443  
333428 ldr.w pc [rD, #imm] instruction leads to assertion  
333501 cachegrind: assertion: Cache set count is not a power of two.  
== 336577  
== 292281  
333666 Recognize MPX instructions and bnd prefix.  
333788 Valgrind does not support the CDROM\_DISC\_STATUS ioctl (has patch)  
333817 Valgrind reports the memory areas written to by the SG\_IO  
ioctl as untouched  
334049 lzcmt fails silently (x86\_32)  
334384 Valgrind does not have support Little Endian support for  
IBM POWER PPC 64  
334585 recvmmsg unhandled (+patch) (arm)  
334705 sendmsg and recvmmsg should guard against bogus msghdr fields.  
334727 Build fails with -Werror=format-security  
334788 clarify doc about --log-file initial program directory  
334834 PPC64 Little Endian support, patch 2  
334836 PPC64 Little Endian support, patch 3 testcase fixes  
334936 patch to fix false positives on alsa SNDRV\_CTL\_\* ioctls  
335034 Unhandled ioctl: HCIGETDEVLIST  
335155 vgdb, fix error print statement.  
335262 arm64: movi 8bit version is not supported  
335263 arm64: dmb instruction is not implemented  
335441 unhandled ioctl 0x8905 (SIOCATMARK) when running wine under valgrind  
335496 arm64: sbc/abc instructions are not implemented  
335554 arm64: unhandled instruction: abs  
335564 arm64: unhandled instruction: fcvtu Xn, Sn  
335735 arm64: unhandled instruction: cnt  
335736 arm64: unhandled instruction: uaddlv  
335848 arm64: unhandled instruction: {s,u}cvtf  
335902 arm64: unhandled instruction: sli  
335903 arm64: unhandled instruction: umull (vector)  
336055 arm64: unhandled instruction: mov (element)  
336062 arm64: unhandled instruction: shrn{,2}  
336139 mip64: [...] valgrind hangs and spins on a single core [...]  
336189 arm64: unhandled Instruction: mvn  
336435 Valgrind hangs in pthread\_spin\_lock consuming 100% CPU  
336619 valgrind --read-var-info=yes doesn't handle DW\_TAG\_restrict\_type  
336772 Make moans about unknown ioctls more informative  
336957 Add a section about the Solaris/illumos port on the webpage  
337094 ifunc wrapper is broken on ppc64  
337285 fcntl commands F\_OFD\_SETLK, F\_OFD\_SETLKW, and F\_OFD\_GETLK not supported  
337528 leak check heuristic for block prefixed by length as 64bit number  
337740 Implement additional Xen hypercalls  
337762 guest\_arm64\_toIR.c:4166 (dis\_ARM64\_load\_store): Assertion `0' failed.  
337766 arm64-linux: unhandled syscalls mlock (228) and mlockall (230)  
337871 deprecate --db-attach  
338023 Add support for all V4L2/media ioctls  
338024 inlined functions are not shown if DW\_AT\_ranges is used  
338106 Add support for 'kcmp' syscall  
338115 DRD: computed conflict set differs from actual after fork  
338160 implement display of thread local storage in gdbdrv  
338205 configure.ac and check for -Wno-tautological-compare  
338300 coredumps are missing one byte of every segment  
338445 amd64 vbit-test fails with unknown opcodes used by arm64 VEX  
338499 --sim-hints parsing broken due to wrong order in tokens  
338615 suppress glibc 2.20 optimized strcmp implementation for ARMv7

338681 Unable to unwind through clone thread created on i386-linux  
 338698 race condition between gdbdrv and vgdb on startup  
 338703 helgrind on arm-linux gets false positives in dynamic loader  
 338791 alt dwz files can be relative of debug/main file  
 338878 on MacOS: assertion 'VG\_IS\_PAGE\_ALIGNED(clstack\_end+1)' failed  
 338932 build V-trunk with gcc-trunk  
 338974 glibc 2.20 changed size of struct sigaction sa\_flags field on s390  
 345079 Fix build problems in VEX/useful/test\_main.c  
 n-i-bz Fix KVM\_CREATE\_IRQCHIP ioctl handling  
 n-i-bz s390x: Fix memory corruption for multithreaded applications  
 n-i-bz vex arm->IR: allow PC as basereg in some LDRD cases  
 n-i-bz internal error in Valgrind if vgdb transmit signals when ptrace invoked  
 n-i-bz Fix mingw64 support in valgrind.h (dev@, 9 May 2014)  
 n-i-bz drd manual: Document how to C++11 programs that use class "std::thread"  
 n-i-bz Add command-line option --default-suppressions  
 n-i-bz Add support for BLKDISCARDZEROES ioctl  
 n-i-bz ppc32/64: fix a regression with the mtfbs0/mtfsb1 instructions  
 n-i-bz Add support for sys\_pivot\_root and sys\_unshare

(3.10.0.BETA1: 2 September 2014, vex r2940, valgrind r14428)  
 (3.10.0.BETA2: 8 September 2014, vex r2950, valgrind r14503)  
 (3.10.0: 10 September 2014, vex r2950, valgrind r14514)

#### Release 3.9.0 (31 October 2013)

~~~~~

3.9.0 is a feature release with many improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, ARM/Android, X86/Android, X86/MacOSX 10.7 and AMD64/MacOSX 10.7. Support for MacOSX 10.8 is significantly improved relative to the 3.8.0 release.

#### \* ===== PLATFORM CHANGES =====

- \* Support for MIPS64 LE and BE running Linux. Valgrind has been tested on MIPS64 Debian Squeeze and Debian Wheezy distributions.
- \* Support for MIPS DSP ASE on MIPS32 platforms.
- \* Support for s390x Decimal Floating Point instructions on hosts that have the DFP facility installed.
- \* Support for POWER8 (Power ISA 2.07) instructions
- \* Support for Intel AVX2 instructions. This is available only on 64 bit code.
- \* Initial support for Intel Transactional Synchronization Extensions, both RTM and HLE.
- \* Initial support for Hardware Transactional Memory on POWER.
- \* Improved support for MacOSX 10.8 (64-bit only). Memcheck can now run large GUI apps tolerably well.

\* ===== TOOL CHANGES =====

\* Memcheck:

- Improvements in handling of vectorised code, leading to significantly fewer false error reports. You need to use the flag `--partial-loads-ok=yes` to get the benefits of these changes.
- Better control over the leak checker. It is now possible to specify which leak kinds (definite/indirect/possible/reachable) should be displayed, which should be regarded as errors, and which should be suppressed by a given leak suppression. This is done using the options `--show-leak-kinds=kind1,kind2,...`, `--errors-for-leak-kinds=kind1,kind2,..` and an optional "match-leak-kinds:" line in suppression entries, respectively.

Note that generated leak suppressions contain this new line and are therefore more specific than in previous releases. To get the same behaviour as previous releases, remove the "match-leak-kinds:" line from generated suppressions before using them.

- Reduced "possible leak" reports from the leak checker by the use of better heuristics. The available heuristics provide detection of valid interior pointers to `std::string`, to `new[]` allocated arrays with elements having destructors and to interior pointers pointing to an inner part of a C++ object using multiple inheritance. They can be selected individually using the option `--leak-check-heuristics=heur1,heur2,...`
- Better control of stacktrace acquisition for heap-allocated blocks. Using the `--keep-stacktraces` option, it is possible to control independently whether a stack trace is acquired for each allocation and deallocation. This can be used to create better "use after free" errors or to decrease Valgrind's resource consumption by recording less information.
- Better reporting of leak suppression usage. The list of used suppressions (shown when the `-v` option is given) now shows, for each leak suppressions, how many blocks and bytes it suppressed during the last leak search.

\* Helgrind:

- False errors resulting from the use of statically initialised mutexes and condition variables (`PTHREAD_MUTEX_INITIALIZER`, etc) have been removed.
- False errors resulting from the use of `pthread_cond_wait`s that timeout, have been removed.

\* ===== OTHER CHANGES =====

\* Some attempt to tune Valgrind's space requirements to the expected capabilities of the target:

- The default size of the translation cache has been reduced from 8 sectors to 6 on Android platforms, since each sector occupies about 40MB when using Memcheck.

- The default size of the translation cache has been increased to 16 sectors on all other platforms, reflecting the fact that large applications require instrumentation and storage of huge amounts of code. For similar reasons, the number of memory mapped segments that can be tracked has been increased by a factor of 6.

- In all cases, the maximum number of sectors in the translation cache can be controlled by the new flag `--num-transtab-sectors`.

\* Changes in how debug info (line numbers, etc) is read:

- Valgrind no longer temporarily mmaps the entire object to read from it. Instead, reading is done through a small fixed sized buffer. This avoids virtual memory usage spikes when Valgrind reads debuginfo from large shared objects.

- A new experimental remote debug info server. Valgrind can read debug info from a different machine (typically, a build host) where debuginfo objects are stored. This can save a lot of time and hassle when running Valgrind on resource-constrained targets (phones, tablets) when the full debuginfo objects are stored somewhere else. This is enabled by the `--debuginfo-server=` option.

- Consistency checking between main and debug objects can be disabled using the `--allow-mismatched-debuginfo` option.

\* Stack unwinding by stack scanning, on ARM. Unwinding by stack scanning can recover stack traces in some cases when the normal unwind mechanisms fail. Stack scanning is best described as "a nasty, dangerous and misleading hack" and so is disabled by default. Use `--unw-stack-scan-thresh` and `--unw-stack-scan-frames` to enable and control it.

\* Detection and merging of recursive stack frame cycles. When your program has recursive algorithms, this limits the memory used by Valgrind for recorded stack traces and avoids recording uninteresting repeated calls. This is controlled by the command line option `--merge-recursive-frame` and by the monitor command `"v.set merge-recursive-frames"`.

\* File name and line numbers for used suppressions. The list of used suppressions (shown when the `-v` option is given) now shows, for each used suppression, the file name and line number where the suppression is defined.

\* New and modified GDB server monitor features:

- `valgrind.h` has a new client request, `VALGRIND_MONITOR_COMMAND`, that can be used to execute gdbserver monitor commands from the client program.

- A new monitor command, `"v.info open_fds"`, that gives the list of open file descriptors and additional details.

- An optional message in the `"v.info n_errs_found"` monitor command, for example `"v.info n_errs_found test 1234 finished"`, allowing a

comment string to be added to the process output, perhaps for the purpose of separating errors of different tests or test phases.

- A new monitor command "v.info execontext" that shows information about the stack traces recorded by Valgrind.

- A new monitor command "v.do expensive\_sanity\_check\_general" to run some internal consistency checks.

\* New flag --sigill-diagnostics to control whether a diagnostic message is printed when the JIT encounters an instruction it can't translate. The actual behavior -- delivery of SIGILL to the application -- is unchanged.

\* The maximum amount of memory that Valgrind can use on 64 bit targets has been increased from 32GB to 64GB. This should make it possible to run applications on Memcheck that natively require up to about 35GB.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)

where XXXXXX is the bug number as listed below.

123837 system call: 4th argument is optional, depending on cmd  
 135425 memcheck should tell you where Freed blocks were Mallocd  
 164485 VG\_N\_SEGNAMEs and VG\_N\_SEGMENTS are (still) too small  
 207815 Adds some of the drm ioctls to syswrap-linux.c  
 251569 vex amd64->IR: 0xF 0x1 0xF9 0xBF 0x90 0xD0 0x3 0x0 (RDTSCP)  
 252955 Impossible to compile with ccache  
 253519 Memcheck reports auxv pointer accesses as invalid reads.  
 263034 Crash when loading some PPC64 binaries  
 269599 Increase deepest backtrace  
 274695 s390x: Support "compare to/from logical" instructions (z196)  
 275800 s390x: Autodetect cache info (part 2)  
 280271 Valgrind reports possible memory leaks on still-reachable std::string  
 284540 Memcheck shouldn't count suppressions matching still-reachable [..]  
 289578 Backtraces with ARM unwind tables (stack scan flags)  
 296311 Wrong stack traces due to -fomit-frame-pointer (x86)  
 304832 ppc32: build failure  
 305431 Use find\_buildid shdr fallback for separate .debug files  
 305728 Add support for AVX2 instructions  
 305948 ppc64: code generation for ShlD64 / ShrD64 asserts  
 306035 s390x: Fix IR generation for LAAG and friends  
 306054 s390x: Condition code computation for convert-to-int/logical  
 306098 s390x: alternate opcode form for convert to/from fixed  
 306587 Fix cache line detection from auxiliary vector for PPC.  
 306783 Mips unhandled syscall : 4025 / 4079 / 4182  
 307038 DWARF2 CFI reader: unhandled DW\_OP\_ opcode 0x8 (DW\_OP\_const1u et al)  
 307082 HG false positive: pthread\_cond\_destroy: destruction of unknown CV  
 307101 sys\_capget second argument can be NULL

307103 sys\_openat: If pathname is absolute, then dirfd is ignored.  
 307106 amd64->IR: f0 0f c0 02 (lock xadd byte)  
 307113 s390x: DFP support  
 307141 valgrind does't work in mips-linux system  
 307155 filter\_gdb should filter out syscall-template.S T\_PSEUDO  
 307285 x86\_amd64 feature test for avx in test suite is wrong  
 307290 memcheck overlap testcase needs memcpy version filter  
 307463 Please add "&limit=0" to the "all open bugs" link  
 307465 --show-possibly-lost=no should reduce the error count / exit code  
 307557 Leaks on Mac OS X 10.7.5 libraries at ImageLoader::recursiveInit[..]  
 307729 pkgconfig support broken valgrind.pc  
 307828 Memcheck false errors SSE optimized wcsncpy, wcsncmp, wcsrchr, wcsrchr  
 307955 Building valgrind 3.7.0-r4 fails in Gentoo AMD64 when using clang  
 308089 Unhandled syscall on ppc64: prctl  
 308135 PPC32 MPC8xx has 16 bytes cache size  
 308321 testsuite memcheck filter interferes with gdb\_filter  
 308333 == 307106  
 308341 vgdb should report process exit (or fatal signal)  
 308427 s390 memcheck reports tsearch cjump/cmove depends on uninit  
 308495 Remove build dependency on installed Xen headers  
 308573 Internal error on 64-bit instruction executed in 32-bit mode  
 308626 == 308627  
 308627 pmovmskb validity bit propagation is imprecise  
 308644 vgdb command for having the info for the track-fds option  
 308711 give more info about aspacemgr and arenas in out\_of\_memory  
 308717 ARM: implement fixed-point VCVT.F64.[SU]32  
 308718 ARM implement SMLALBB family of instructions  
 308886 Missing support for PTRACE\_SET/GETREGSET  
 308930 syscall name\_to\_handle\_at (303 on amd64) not handled  
 309229 V-bit tester does not report number of tests generated  
 309323 print unrecognized instruction on MIPS  
 309425 Provide a --sigill-diagnostics flag to suppress illegal [..]  
 309427 SSE optimized stpncpy trigger uninitialised value [..] errors  
 309430 Self hosting ppc64 encounters a vassert error on operand type  
 309600 valgrind is a bit confused about 0-sized sections  
 309823 Generate errors for still reachable blocks  
 309921 PCMPISTRI validity bit propagation is imprecise  
 309922 none/tests/ppc64/test\_dfp5 sometimes fails  
 310169 The Iop\_CmpORD class of Iops is not supported by the vbit checker.  
 310424 --read-var-info does not properly describe static variables  
 310792 search additional path for debug symbols  
 310931 s390x: Message-security assist (MSA) instruction extension [..]  
 311100 PPC DFP implementation of the integer operands is inconsistent [..]  
 311318 ARM: "128-bit constant is not implemented" error message  
 311407 ssse3 bcopy (actually converted memcpy) causes invalid read [..]  
 311690 V crashes because it redirects branches inside of a redirected function  
 311880 x86\_64: make regtest hangs at shell\_valid1  
 311922 WARNING: unhandled syscall: 170  
 311933 == 251569  
 312171 ppc: insn selection for DFP  
 312571 Rounding mode call wrong for the DFP Iops [..]  
 312620 Change to Iop\_D32toD64 [..] for s390 DFP support broke ppc [..]  
 312913 Dangling pointers error should also report the alloc stack trace  
 312980 Building on Mountain Lion generates some compiler warnings  
 313267 Adding MIPS64/Linux port to Valgrind  
 313348 == 251569  
 313354 == 251569  
 313811 Buffer overflow in assert\_fail

314099 coverity pointed out error in VEX guest\_ppc\_toIR.c insn\_suffix  
314269 ppc: dead code in insn selection  
314718 ARM: implement integer divide instruction (sdiv and udiv)  
315345 cl-format.xml and callgrind/dump.c don't agree on using cfl= or cfi=  
315441 sendmsg syscall should ignore unset msghdr msg\_flags  
315534 msgrcv inside a thread causes valgrind to hang (block)  
315545 Assertion '(UChar\*)sec->tt[tteNo].tcptr <= (UChar\*)hcode' failed  
315689 disInstr(thumb): unhandled instruction: 0xF852 0x0E10 (LDRT)  
315738 disInstr(arm): unhandled instruction: 0xEEBE0BEE (vcvt.s32.f64)  
315959 valgrind man page has bogus SGCHECK (and no BBV) OPTIONS section  
316144 valgrind.1 manpage contains unknown ??? strings [..]  
316145 callgrind command line options in manpage reference (unknown) [..]  
316145 callgrind command line options in manpage reference [..]  
316181 drd: Fixed a 4x slowdown for certain applications  
316503 Valgrind does not support SSE4 "movntdqa" instruction  
316535 Use of |signed int| instead of |size\_t| in valgrind messages  
316696 fluidanimate program of parsec 2.1 stuck  
316761 syscall open\_by\_handle\_at (304 on amd64, 342 on x86) not handled  
317091 Use -Wl,-Ttext-segment when static linking if possible [..]  
317186 "Impossible happens" when occurs VCVT instruction on ARM  
317318 Support for Threading Building Blocks "scalable\_malloc"  
317444 amd64->IR: 0xC4 0x41 0x2C 0xC2 0xD2 0x8 (vcmpsq\_uqps)  
317461 Fix BMI assembler configure check and avx2/bmi/fma vgtest prereqs  
317463 bmi testcase IR SANITY CHECK FAILURE  
317506 memcheck/tests/vbit-test fails with unknown opcode after [..]  
318050 libmpiwrap fails to compile with out-of-source build  
318203 setsockopt handling needs to handle SOL\_SOCKET/SO\_ATTACH\_FILTER  
318643 annotate\_trace\_memory tests infinite loop on arm and ppc [..]  
318773 amd64->IR: 0xF3 0x48 0x0F 0xBC 0xC2 0xC3 0x66 0x0F  
318929 Crash with: disInstr(thumb): 0xF321 0x0001 (ssat16)  
318932 Add missing PPC64 and PPC32 system call support  
319235 --db-attach=yes is broken with Yama (ptrace scoping) enabled  
319395 Crash with unhandled instruction on STRT (Thumb) instructions  
319494 VEX Makefile-gcc standalone build update after r2702  
319505 [MIPSEL] Crash: unhandled UNRAY operator.  
319858 disInstr(thumb): unhandled instruction on instruction STRBT  
319932 disInstr(thumb): unhandled instruction on instruction STRHT  
320057 Problems when we try to mmap more than 12 memory pages on MIPS32  
320063 Memory from PTRACE\_GET\_THREAD\_AREA is reported uninitialised  
320083 disInstr(thumb): unhandled instruction on instruction LDRBT  
320116 bind on AF\_BLUETOOTH produces warnings because of sockaddr\_rc padding  
320131 WARNING: unhandled syscall: 369 on ARM (prlimit64)  
320211 Stack buffer overflow in ./coregrind/m\_main.c with huge TMPDIR  
320661 vgModuleLocal\_read\_elf\_debug\_info(): "Assertion '!di->soname'  
320895 add fanotify support (patch included)  
320998 vex amd64->IR pcmpestri and pcmpestrm SSE4.2 instruction  
321065 Valgrind updates for Xen 4.3  
321148 Unhandled instruction: PLI (Thumb 1, 2, 3)  
321363 Unhandled instruction: SSAX (ARM + Thumb)  
321364 Unhandled instruction: SXTAB16 (ARM + Thumb)  
321466 Unhandled instruction: SHASX (ARM + Thumb)  
321467 Unhandled instruction: SHSAX (ARM + Thumb)  
321468 Unhandled instruction: SHSUB16 (ARM + Thumb)  
321619 Unhandled instruction: SHSUB8 (ARM + Thumb)  
321620 Unhandled instruction: UASX (ARM + Thumb)  
321621 Unhandled instruction: USAX (ARM + Thumb)  
321692 Unhandled instruction: UQADD16 (ARM + Thumb)  
321693 Unhandled instruction: LDRSBT (Thumb)

321694 Unhandled instruction: UQASX (ARM + Thumb)  
 321696 Unhandled instruction: UQSAX (Thumb + ARM)  
 321697 Unhandled instruction: UHASX (ARM + Thumb)  
 321703 Unhandled instruction: UHSAX (ARM + Thumb)  
 321704 Unhandled instruction: REVSH (ARM + Thumb)  
 321730 Add cg\_diff and cg\_merge man pages  
 321738 Add vgdb and valgrind-listener man pages  
 321814 == 315545  
 321891 Unhandled instruction: LDRHT (Thumb)  
 321960 pthread\_create() then alloca() causing invalid stack write errors  
 321969 ppc32 and ppc64 don't support [lf]setxattr  
 322254 Show threadname together with tid if set by application  
 322294 Add initial support for IBM Power ISA 2.07  
 322368 Assertion failure in wqthread\_hijack under OS X 10.8  
 322563 vex mips->IR: 0x70 0x83 0xF0 0x3A  
 322807 VALGRIND\_PRINTF\_BACKTRACE writes callstack to xml and text to stderr  
 322851 0bXXX binary literal syntax is not standard  
 323035 Unhandled instruction: LDRSHT(Thumb)  
 323036 Unhandled instruction: SMMLS (ARM and Thumb)  
 323116 The memcheck/tests/ppc64/power\_ISA2\_05.c fails to build [..]  
 323175 Unhandled instruction: SMLALD (ARM + Thumb)  
 323177 Unhandled instruction: SMLSLD (ARM + Thumb)  
 323432 Calling pthread\_cond\_destroy() or pthread\_mutex\_destroy() [..]  
 323437 Phase 2 support for IBM Power ISA 2.07  
 323713 Support mmxext (integer sse) subset on i386 (athlon)  
 323803 Transactional memory instructions are not supported for Power  
 323893 SSE3 not available on amd cpus in valgrind  
 323905 Probable false positive from Valgrind/drd on close()  
 323912 valgrind.h header isn't compatible for mingw64  
 324047 Valgrind doesn't support [LDR,ST]{S}[B,H]T ARM instructions  
 324149 helgrind: When pthread\_cond\_timedwait returns ETIMEDOUT [..]  
 324181 mmap does not handle MAP\_32BIT  
 324227 memcheck false positive leak when a thread calls exit+block [..]  
 324421 Support for fanotify API on ARM architecture  
 324514 gdbserver monitor cmd output behaviour consistency [..]  
 324518 ppc64: Emulation of dcbt instructions does not handle [..]  
 324546 none/tests/ppc32 test\_isa\_2\_07\_part2 requests -m64  
 324582 When access is made to freed memory, report both allocation [..]  
 324594 Fix overflow computation for Power ISA 2.06 insns: mulldo/mulldo.  
 324765 ppc64: illegal instruction when executing none/tests/ppc64/jm-misc  
 324816 Incorrect VEX implementation for xscvspdp/xvcvspdp for SNaN inputs  
 324834 Unhandled instructions in Microsoft C run-time for x86\_64  
 324894 Phase 3 support for IBM Power ISA 2.07  
 326091 drd: Avoid false race reports from optimized strlen() impls  
 326113 valgrind libvex hwcaps error on AMD64  
 n-i-bz Some wrong command line options could be ignored  
 n-i-bz patch to allow fair-sched on android  
 n-i-bz report error for vgdb snapshot requested before execution  
 n-i-bz same as 303624 (fixed in 3.8.0), but for x86 android

(3.9.0: 31 October 2013, vex r2796, valgrind r13708)

Release 3.8.1 (19 September 2012)

~~~~~

3.8.1 is a bug fix release. It fixes some assertion failures in 3.8.0 that occur moderately frequently in real use cases, adds support for

some missing instructions on ARM, and fixes a deadlock condition on MacOSX. If you package or deliver 3.8.0 for others to use, you might want to consider upgrading to 3.8.1 instead.

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)

where XXXXXX is the bug number as listed below.

284004 == 301281  
 289584 Unhandled instruction: 0xF 0x29 0xE5 (MOVAPS)  
 295808 amd64->IR: 0xF3 0xF 0xBC 0xC0 (TZCNT)  
 298281 wcslen causes false(?) uninitialised value warnings  
 301281 valgrind hangs on OS X when the process calls system()  
 304035 disInstr(arm): unhandled instruction 0xE1023053  
 304867 implement MOVBE instruction in x86 mode  
 304980 Assertion 'lo <= hi' failed in vgModuleLocal\_find\_rx\_mapping  
 305042 amd64: implement 0F 7F encoding of movq between two registers  
 305199 ARM: implement QDADD and QDSUB  
 305321 amd64->IR: 0xF 0xD 0xC (prefetchw)  
 305513 killed by fatal signal: SIGSEGV  
 305690 DRD reporting invalid semaphore when sem\_trywait fails  
 305926 Invalid alignment checks for some AVX instructions  
 306297 disInstr(thumb): unhandled instruction 0xE883 0x000C  
 306310 3.8.0 release tarball missing some files  
 306612 RHEL 6 glibc-2.X default suppressions need /lib\*/libc-\*patterns  
 306664 vex amd64->IR: 0x66 0xF 0x3A 0x62 0xD1 0x46 0x66 0xF  
 n-i-bz shmat of a segment > 4Gb does not work  
 n-i-bz simulate\_control\_c script wrong USR1 signal number on mips  
 n-i-bz vgdb ptrace calls wrong on mips [...]  
 n-i-bz Fixes for more MPI false positives  
 n-i-bz exp-sgcheck's memcpy causes programs to segfault  
 n-i-bz OSX build w/ clang: asserts at startup  
 n-i-bz Incorrect undef'dness prop for Iop\_DPBtoBCD and Iop\_BCDtoDPB  
 n-i-bz fix a couple of union tag-vs-field mixups  
 n-i-bz OSX: use \_\_NR\_poll\_nocancel rather than \_\_NR\_poll

The following bugs were fixed in 3.8.0 but not listed in this NEWS file at the time:

254088 Valgrind should know about UD2 instruction  
 301280 == 254088  
 301902 == 254088  
 304754 NEWS blows TeX's little mind

(3.8.1: 19 September 2012, vex r2537, valgrind r12996)

Release 3.8.0 (10 August 2012)

~~~~~

3.8.0 is a feature release with many improvements and the usual

collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, MIPS/Linux, ARM/Android, X86/Android, X86/MacOSX 10.6/10.7 and AMD64/MacOSX 10.6/10.7. Support for recent distros and toolchain components (glibc 2.16, gcc 4.7) has been added. There is initial support for MacOSX 10.8, but it is not usable for serious work at present.

\* ===== PLATFORM CHANGES =====

\* Support for MIPS32 platforms running Linux. Valgrind has been tested on MIPS32 and MIPS32r2 platforms running different Debian Squeeze and MeeGo distributions. Both little-endian and big-endian cores are supported. The tools Memcheck, Massif and Lackey have been tested and are known to work. See README.mips for more details.

\* Preliminary support for Android running on x86.

\* Preliminary (as-yet largely unusable) support for MacOSX 10.8.

\* Support for Intel AVX instructions and for AES instructions. This support is available only for 64 bit code.

\* Support for POWER Decimal Floating Point instructions.

\* ===== TOOL CHANGES =====

\* Non-libc malloc implementations are now supported. This is useful for tools that replace malloc (Memcheck, Massif, DRD, Helgrind). Using the new option --soname-synonyms, such tools can be informed that the malloc implementation is either linked statically into the executable, or is present in some other shared library different from libc.so. This makes it possible to process statically linked programs, and programs using other malloc libraries, for example TCMalloc or JEMalloc.

\* For tools that provide their own replacement for malloc et al, the option --redzone-size=<number> allows users to specify the size of the padding blocks (redzones) added before and after each client allocated block. Smaller redzones decrease the memory needed by Valgrind. Bigger redzones increase the chance to detect blocks overrun or underrun. Prior to this change, the redzone size was hardwired to 16 bytes in Memcheck.

\* Memcheck:

- The leak\_check GDB server monitor command now can control the maximum nr of loss records to output.
- Reduction of memory use for applications allocating many blocks and/or having many partially defined bytes.
- Addition of GDB server monitor command 'block\_list' that lists the addresses/sizes of the blocks of a leak search loss record.
- Addition of GDB server monitor command 'who\_points\_at' that lists the locations pointing at a block.

- If a redzone size  $> 0$  is given, `VALGRIND_MALLOCLIKE_BLOCK` now will detect an invalid access of these redzones, by marking them `noaccess`. Similarly, if a redzone size is given for a memory pool, `VALGRIND_MEMPOOL_ALLOC` will mark the redzones `no access`. This still allows to find some bugs if the user has forgotten to mark the pool superblock `noaccess`.

- Performance of memory leak check has been improved, especially in cases where there are many leaked blocks and/or many suppression rules used to suppress leak reports.

- Reduced noise (false positive) level on MacOSX 10.6/10.7, due to more precise analysis, which is important for LLVM/Clang generated code. This is at the cost of somewhat reduced performance. Note there is no change to analysis precision or costs on Linux targets.

\* DRD:

- Added even more facilities that can help finding the cause of a data race, namely the command-line option `--ptrace-addr` and the macro `DRD_STOP_TRACING_VAR(x)`. More information can be found in the manual.

- Fixed a subtle bug that could cause false positive data race reports.

\* ===== OTHER CHANGES =====

- \* The C++ demangler has been updated so as to work well with C++ compiled by up to at least g++ 4.6.

- \* Tool developers can make replacement/wrapping more flexible thanks to the new option `--soname-synonyms`. This was reported above, but in fact is very general and applies to all function replacement/wrapping, not just to malloc-family functions.

- \* Round-robin scheduling of threads can be selected, using the new option `--fair-sched=yes`. Prior to this change, the pipe-based thread serialisation mechanism (which is still the default) could give very unfair scheduling. `--fair-sched=yes` improves responsiveness of interactive multithreaded applications, and improves repeatability of results from the thread checkers Helgrind and DRD.

- \* For tool developers: support to run Valgrind on Valgrind has been improved. We can now routinely Valgrind on Helgrind or Memcheck.

- \* `gdbserver` now shows the float shadow registers as integer rather than float values, as the shadow values are mostly used as bit patterns.

- \* Increased limit for the `--num-callers` command line flag to 500.

- \* Performance improvements for error matching when there are many suppression records in use.

- \* Improved support for DWARF4 debugging information (bug 284184).

\* Initial support for DWZ compressed Dwarf debug info.

\* Improved control over the IR optimiser's handling of the tradeoff between performance and precision of exceptions. Specifically, --vex-iropt-precise-memory-exns has been removed and replaced by --vex-iropt-register-updates, with extended functionality. This allows the Valgrind gdbserver to always show up to date register values to GDB.

\* Modest performance gains through the use of translation chaining for JIT-generated code.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([https://bugs.kde.org/enter\\_bug.cgi?product=valgrind](https://bugs.kde.org/enter_bug.cgi?product=valgrind)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit

[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
where XXXXXX is the bug number as listed below.

197914 Building valgrind from svn now requires automake-1.10  
203877 increase to 16Mb maximum allowed alignment for memalign et al  
219156 Handle statically linked malloc or other malloc lib (e.g. tcmalloc)  
247386 make perf does not run all performance tests  
270006 Valgrind scheduler unfair  
270777 Adding MIPS/Linux port to Valgrind  
270796 s390x: Removed broken support for the TS insn  
271438 Fix configure for proper SSE4.2 detection  
273114 s390x: Support TR, TRE, TROO, TROT, TRTO, and TRTT instructions  
273475 Add support for AVX instructions  
274078 improved configure logic for mpicc  
276993 fix mremap 'no thrash checks'  
278313 Fedora 15/x64: err read debug info with --read-var-info=yes flag  
281482 memcheck incorrect byte allocation count in realloc() for silly argument  
282230 group allocator for small fixed size, use it for MC\_Chunk/SEc vbit  
283413 Fix wrong sanity check  
283671 Robustize alignment computation in LibVEX\_Alloc  
283961 Adding support for some HCI IOCTLs  
284124 parse\_type\_DIE: confused by: DWARF 4  
284864 == 273475 (Add support for AVX instructions)  
285219 Too-restrictive constraints for Thumb2 "SP plus/minus register"  
285662 (MacOSX): Memcheck needs to replace memcpy/memmove  
285725 == 273475 (Add support for AVX instructions)  
286261 add wrapper for linux I2C\_RDWR ioctl  
286270 vgpreload is not friendly to 64->32 bit execs, gives ld.so warnings  
286374 Running cachegrind with --branch-sim=yes on 64-bit PowerPC program fails  
286384 configure fails "checking for a supported version of gcc"  
286497 == 273475 (Add support for AVX instructions)  
286596 == 273475 (Add support for AVX instructions)  
286917 disInstr(arm): unhandled instruction: QADD (also QSUB)  
287175 ARM: scalar VFP fixed-point VCVT instructions not handled  
287260 Incorrect conditional jump or move depends on uninitialised value(s)  
287301 vex amd64->IR: 0x66 0xF 0x38 0x41 0xC0 0xB8 0x0 0x0 (PHMINPOSUW)

287307 == 273475 (Add support for AVX instructions)  
 287858 VG\_(strerror): unknown error  
 288298 (MacOSX) unhandled syscall shm\_unlink  
 288995 == 273475 (Add support for AVX instructions)  
 289470 Loading of large Mach-O thin binaries fails.  
 289656 == 273475 (Add support for AVX instructions)  
 289699 vgdb connection in relay mode erroneously closed due to buffer overrun  
 289823 == 293754 (PCMPxSTRx not implemented for 16-bit characters)  
 289839 s390x: Provide support for unicode conversion instructions  
 289939 monitor cmd 'leak\_check' with details about leaked or reachable blocks  
 290006 memcheck doesn't mark %xmm as initialized after "pcmpeqw %xmm %xmm"  
 290655 Add support for AESKEYGENASSIST instruction  
 290719 valgrind-3.7.0 fails with automake-1.11.2 due to "pkglibdir" usage  
 290974 vgdb must align pages to VKI\_SHMLBA (16KB) on ARM  
 291253 ES register not initialised in valgrind simulation  
 291568 Fix 3DNOW-related crashes with baseline x86\_64 CPU (w patch)  
 291865 s390x: Support the "Compare Double and Swap" family of instructions  
 292300 == 273475 (Add support for AVX instructions)  
 292430 unrecognized instruction in \_\_intel\_get\_new\_mem\_ops\_cpuid  
 292493 == 273475 (Add support for AVX instructions)  
 292626 Missing fcntl F\_SETOWN\_EX and F\_GETOWN\_EX support  
 292627 Missing support for some SCSI ioctls  
 292628 none/tests/x86/bug125959-x86.c triggers undefined behavior  
 292841 == 273475 (Add support for AVX instructions)  
 292993 implement the getcpu syscall on amd64-linux  
 292995 Implement the "cross memory attach" syscalls introduced in Linux 3.2  
 293088 Add some VEX sanity checks for ppc64 unhandled instructions  
 293751 == 290655 (Add support for AESKEYGENASSIST instruction)  
 293754 PCMPxSTRx not implemented for 16-bit characters  
 293755 == 293754 (No tests for PCMPxSTRx on 16-bit characters)  
 293808 CLFLUSH not supported by latest VEX for amd64  
 294047 valgrind does not correctly emulate prlimit64(..., RLIMIT\_NOFILE, ...)  
 294048 MPSADBW instruction not implemented  
 294055 regtest none/tests/shell fails when locale is not set to C  
 294185 INT 0x44 (and others) not supported on x86 guest, but used by Jikes RVM  
 294190 --vgdb-error=xxx can be out of sync with errors shown to the user  
 294191 amd64: fnsave/frstor and 0x66 size prefixes on FP instructions  
 294260 disInstr\_AMD64: disInstr miscalculated next %rip  
 294523 --partial-loads-ok=yes causes false negatives  
 294617 vex amd64->IR: 0x66 0xF 0x3A 0xDF 0xD1 0x1 0xE8 0x6A  
 294736 vex amd64->IR: 0x48 0xF 0xD7 0xD6 0x48 0x83  
 294812 patch allowing to run (on x86 at least) helgrind/drd on tool.  
 295089 can not annotate source for both helgrind and drd  
 295221 POWER Processor decimal floating point instruction support missing  
 295427 building for i386 with clang on darwin11 requires "-new\_linker linker"  
 295428 coregrind/m\_main.c has incorrect x86 assembly for darwin  
 295590 Helgrind: Assertion 'cvi->nWaiters > 0' failed  
 295617 ARM - Add some missing syscalls  
 295799 Missing \n with get\_vbits in gdbserver when line is % 80 [...]  
 296229 Linux user input device ioctls missing wrappers  
 296318 ELF Debug info improvements (more than one rx/rw mapping)  
 296422 Add translation chaining support  
 296457 vex amd64->IR: 0x66 0xF 0x3A 0xDF 0xD1 0x1 0xE8 0x6A (dup of AES)  
 296792 valgrind 3.7.0: add SIOCSHWTSTAMP (0x89B0) ioctl wrapper  
 296983 Fix build issues on x86\_64/ppc64 without 32-bit toolchains  
 297078 gdbserver signal handling problems [...]  
 297147 drd false positives on newly allocated memory  
 297329 disallow decoding of IBM Power DFP insns on some machines

297497 POWER Processor decimal floating point instruction support missing  
 297701 Another alias for strncasecmp\_1 in libc-2.13.so  
 297911 'invalid write' not reported when using APIs for custom mem allocators.  
 297976 s390x: revisit EX implementation  
 297991 Valgrind interferes with mmap()+ftell()  
 297992 Support systems missing WIFCONTINUED (e.g. pre-2.6.10 Linux)  
 297993 Fix compilation of valgrind with gcc -g3.  
 298080 POWER Processor DFP support missing, part 3  
 298227 == 273475 (Add support for AVX instructions)  
 298335 == 273475 (Add support for AVX instructions)  
 298354 Unhandled ARM Thumb instruction 0xEB0D 0x0585 (streq)  
 298394 s390x: Don't bail out on an unknown machine model. [...]  
 298421 accept4() syscall (366) support is missing for ARM  
 298718 vex amd64->IR: 0xF 0xB1 0xCB 0x9C 0x8F 0x45  
 298732 valgrind installation problem in ubuntu with kernel version 3.x  
 298862 POWER Processor DFP instruction support missing, part 4  
 298864 DWARF reader mis-parses DW\_FORM\_ref\_addr  
 298943 massif asserts with --pages-as-heap=yes when brk is changing [...]  
 299053 Support DWARF4 DW\_AT\_high\_pc constant form  
 299104 == 273475 (Add support for AVX instructions)  
 299316 Helgrind: hg\_main.c:628 (map\_threads\_lookup): Assertion 'thr' failed.  
 299629 dup3() syscall (358) support is missing for ARM  
 299694 POWER Processor DFP instruction support missing, part 5  
 299756 Ignore --free-fill for MEMPOOL\_FREE and FREELIKE client requests  
 299803 == 273475 (Add support for AVX instructions)  
 299804 == 273475 (Add support for AVX instructions)  
 299805 == 273475 (Add support for AVX instructions)  
 300140 ARM - Missing (T1) SMMUL  
 300195 == 296318 (ELF Debug info improvements (more than one rx/rw mapping))  
 300389 Assertion 'are\_valid\_hwcaps(VexArchAMD64, [...])' failed.  
 300414 FCOM and FCOMP unimplemented for amd64 guest  
 301204 infinite loop in canonicaliseSymtab with ifunc symbol  
 301229 == 203877 (increase to 16Mb maximum allowed alignment for memalign etc)  
 301265 add x86 support to Android build  
 301984 configure script doesn't detect certain versions of clang  
 302205 Fix compiler warnings for POWER VEX code and POWER test cases  
 302287 Unhandled movbe instruction on Atom processors  
 302370 PPC: fnmadd, fnmsub, fnmadds, fnmsubs insns always negate the result  
 302536 Fix for the POWER Valgrind regression test: memcheck-ISA2.0.  
 302578 Unrecognized instruction 0xc5 0x32 0xc2 0xca 0x09 vcmpngess  
 302656 == 273475 (Add support for AVX instructions)  
 302709 valgrind for ARM needs extra tls support for android emulator [...]  
 302827 add wrapper for CDROM\_GET\_CAPABILITY  
 302901 Valgrind crashes with dwz optimized debuginfo  
 302918 Enable testing of the vmaddfp and vnsbf instructions in the testsuite  
 303116 Add support for the POWER instruction popcntb  
 303127 Power test suite fixes for frsqtrte, vrefp, and vrsqrtefp instructions.  
 303250 Assertion 'instrs\_in->arr\_used <= 10000' failed w/ OpenSSL code  
 303466 == 273475 (Add support for AVX instructions)  
 303624 segmentation fault on Android 4.1 (e.g. on Galaxy Nexus OMAP)  
 303963 strstr() function produces wrong results under valgrind callgrind  
 304054 CALL\_FN\_xx macros need to enforce stack alignment  
 304561 tee system call not supported  
 715750 (MacOSX): Incorrect invalid-address errors near 0xFFFFxxxx (mozbug#)  
 n-i-bz Add missing gdbserver xml files for shadow registers for ppc32  
 n-i-bz Bypass gcc4.4/4.5 code gen bugs causing out of memory or asserts  
 n-i-bz Fix assert in gdbserver for watchpoints watching the same address  
 n-i-bz Fix false positive in sys\_clone on amd64 when optional args [...]

n-i-bz s390x: Shadow registers can now be examined using vgdb

(3.8.0-TEST3: 9 August 2012, vex r2465, valgrind r12865)

(3.8.0: 10 August 2012, vex r2465, valgrind r12866)

Release 3.7.0 (5 November 2011)

~~~~~

3.7.0 is a feature release with many significant improvements and the usual collection of bug fixes.

This release supports X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, ARM/Android, X86/Darwin and AMD64/Darwin. Support for recent distros and toolchain components (glibc 2.14, gcc 4.6, MacOSX 10.7) has been added.

\* ===== PLATFORM CHANGES =====

\* Support for IBM z/Architecture (s390x) running Linux. Valgrind can analyse 64-bit programs running on z/Architecture. Most user space instructions up to and including z10 are supported. Valgrind has been tested extensively on z9, z10, and z196 machines running SLES 10/11, RedHat 5/6m, and Fedora. The Memcheck and Massif tools are known to work well. Callgrind, Helgrind, and DRD work reasonably well on z9 and later models. See README.s390 for more details.

\* Preliminary support for MacOSX 10.7 and XCode 4. Both 32- and 64-bit processes are supported. Some complex threaded applications (Firefox) are observed to hang when run as 32 bit applications, whereas 64-bit versions run OK. The cause is unknown. Memcheck will likely report some false errors. In general, expect some rough spots. This release also supports MacOSX 10.6, but drops support for 10.5.

\* Preliminary support for Android (on ARM). Valgrind can now run large applications (eg, Firefox) on (eg) a Samsung Nexus S. See README.android for more details, plus instructions on how to get started.

\* Support for the IBM Power ISA 2.06 (Power7 instructions)

\* General correctness and performance improvements for ARM/Linux, and, by extension, ARM/Android.

\* Further solidification of support for SSE 4.2 in 64-bit mode. AVX instruction set support is under development but is not available in this release.

\* Support for AIX5 has been removed.

\* ===== TOOL CHANGES =====

\* Memcheck: some incremental changes:

- reduction of memory use in some circumstances
- improved handling of freed memory, which in some circumstances

can cause detection of use-after-free that would previously have been missed

- fix of a longstanding bug that could cause false negatives (missed errors) in programs doing vector saturated narrowing instructions.

\* Helgrind: performance improvements and major memory use reductions, particularly for large, long running applications which perform many synchronisation (lock, unlock, etc) events. Plus many smaller changes:

- display of locksets for both threads involved in a race
- general improvements in formatting/clarity of error messages
- addition of facilities and documentation regarding annotation of thread safe reference counted C++ classes
- new flag `--check-stack-refs=no|yes [yes]`, to disable race checking on thread stacks (a performance hack)
- new flag `--free-is-write=no|yes [no]`, to enable detection of races where one thread accesses heap memory but another one frees it, without any coordinating synchronisation event

\* DRD: enabled XML output; added support for delayed thread deletion in order to detect races that occur close to the end of a thread (`--join-list-vol`); fixed a memory leak triggered by repeated client memory allocation and deallocation; improved Darwin support.

\* `exp-ptrcheck`: this tool has been renamed to `exp-sgcheck`

\* `exp-sgcheck`: this tool has been reduced in scope so as to improve performance and remove checking that Memcheck does better. Specifically, the ability to check for overruns for stack and global arrays is unchanged, but the ability to check for overruns of heap blocks has been removed. The tool has accordingly been renamed to `exp-sgcheck` ("Stack and Global Array Checking").

\* ===== OTHER CHANGES =====

\* GDB server: Valgrind now has an embedded GDB server. That means it is possible to control a Valgrind run from GDB, doing all the usual things that GDB can do (single stepping, breakpoints, examining data, etc). Tool-specific functionality is also available. For example, it is possible to query the definedness state of variables or memory from within GDB when running Memcheck; arbitrarily large memory watchpoints are supported, etc. To use the GDB server, start Valgrind with the flag `--vgdb-error=0` and follow the on-screen instructions.

\* Improved support for unfriendly self-modifying code: a new option `--smc-check=all-non-file` is available. This adds the relevant consistency checks only to code that originates in non-file-backed mappings. In effect this confines the consistency checking only to code that is or might be JIT generated, and avoids checks on code that must have been compiled ahead of time. This significantly improves performance on applications that generate code at run time.

\* It is now possible to build a working Valgrind using Clang-2.9 on Linux.

\* new client requests VALGRIND\_{DISABLE,ENABLE}\_ERROR\_REPORTING. These enable and disable error reporting on a per-thread, and nestable, basis. This is useful for hiding errors in particularly troublesome pieces of code. The MPI wrapper library (libmpiwrap.c) now uses this facility.

\* Added the --mod-funcname option to cg\_diff.

\* ===== FIXED BUGS =====

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([http://bugs.kde.org/enter\\_valgrind\\_bug.cgi](http://bugs.kde.org/enter_valgrind_bug.cgi)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit  
[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed below.

79311 malloc silly arg warning does not give stack trace  
 210935 port valgrind.h (not valgrind) to win32 to support client requests  
 214223 valgrind SIGSEGV on startup gcc 4.4.1 ppc32 (G4) Ubuntu 9.10  
 243404 Port to zSeries  
 243935 Helgrind: incorrect handling of ANNOTATE\_HAPPENS\_BEFORE()/AFTER()  
 247223 non-x86: Suppress warning: 'regparm' attribute directive ignored  
 250101 huge "free" memory usage due to m\_mallocfree.c fragmentation  
 253206 Some fixes for the faultstatus testcase  
 255223 capget testcase fails when running as root  
 256703 xlc\_dbl\_u32.c testcase broken  
 256726 Helgrind tests have broken inline asm  
 259977 == 214223 (Valgrind segfaults doing \_\_builtin\_longjmp)  
 264800 testcase compile failure on zseries  
 265762 make public VEX headers compilable by G++ 3.x  
 265771 assertion in jumps.c (r11523) fails with glibc-2.3  
 266753 configure script does not give the user the option to not use QtCore  
 266931 gen\_insn\_test.pl is broken  
 266961 ld-linux.so.2 i?86-linux strlen issues  
 266990 setns instruction causes false positive  
 267020 Make directory for temporary files configurable at run-time.  
 267342 == 267997 (segmentation fault on Mac OS 10.6)  
 267383 Assertion 'vgPlain\_strlen(dir) + vgPlain\_strlen(file) + 1 < 256' failed  
 267413 Assertion 'DRD\_(g\_threadinfo)[tid].synchr\_nesting >= 1' failed.  
 267488 regtest: darwin support for 64-bit build  
 267552 SIGSEGV (misaligned\_stack\_error) with DRD, but not with other tools  
 267630 Add support for IBM Power ISA 2.06 -- stage 1  
 267769 == 267997 (Darwin: memcheck triggers segmentation fault)  
 267819 Add client request for informing the core about reallocation  
 267925 laog data structure quadratic for a single sequence of lock  
 267968 drd: (vgDrd\_thread\_set\_joinable): Assertion '0 <= (int)tid ..' failed  
 267997 MacOSX: 64-bit V segfaults on launch when built with Xcode 4.0.1  
 268513 missed optimizations in fold\_Expr  
 268619 s390x: fpr - gpr transfer facility

268620 s390x: reconsider "long displacement" requirement  
 268621 s390x: improve IR generation for XC  
 268715 s390x: FLOGR is not universally available  
 268792 == 267997 (valgrind seg faults on startup when compiled with Xcode 4)  
 268930 s390x: MHY is not universally available  
 269078 arm->IR: unhandled instruction SUB (SP minus immediate/register)  
 269079 Support ptrace system call on ARM  
 269144 missing "Bad option" error message  
 269209 conditional load and store facility (z196)  
 269354 Shift by zero on x86 can incorrectly clobber CC\_NDEP  
 269641 == 267997 (valgrind segfaults immediately (segmentation fault))  
 269736 s390x: minor code generation tweaks  
 269778 == 272986 (valgrind.h: swap roles of VALGRIND\_DO\_CLIENT\_REQUEST() ..)  
 269863 s390x: remove unused function parameters  
 269864 s390x: tweak s390\_emit\_load\_cc  
 269884 == 250101 (overhead for huge blocks exhausts space too soon)  
 270082 s390x: Make sure to point the PSW address to the next address on SIGILL  
 270115 s390x: rewrite some testcases  
 270309 == 267997 (valgrind crash on startup)  
 270320 add support for Linux FIOQSIZE ioctl() call  
 270326 segfault while trying to sanitize the environment passed to execle  
 270794 IBM POWER7 support patch causes regression in none/tests  
 270851 IBM POWER7 fcfidus instruction causes memcheck to fail  
 270856 IBM POWER7 xsnmaddadp instruction causes memcheck to fail on 32bit app  
 270925 hyper-optimized strspn() in /lib64/libc-2.13.so needs fix  
 270959 s390x: invalid use of R0 as base register  
 271042 VSX configure check fails when it should not  
 271043 Valgrind build fails with assembler error on ppc64 with binutils 2.21  
 271259 s390x: fix code confusion  
 271337 == 267997 (Valgrind segfaults on MacOS X)  
 271385 s390x: Implement Ist\_MBE  
 271501 s390x: misc cleanups  
 271504 s390x: promote likely and unlikely  
 271579 ppc: using wrong enum type  
 271615 unhandled instruction "popcnt" (arch=amd10h)  
 271730 Fix bug when checking ioctls: duplicate check  
 271776 s390x: provide STFLE instruction support  
 271779 s390x: provide clock instructions like STCK  
 271799 Darwin: ioctls without an arg report a memory error  
 271820 arm: fix type confusion  
 271917 pthread\_cond\_timedwait failure leads to not-locked false positive  
 272067 s390x: fix DISP20 macro  
 272615 A typo in debug output in mc\_leakcheck.c  
 272661 callgrind\_annotate chokes when run from paths containing regex chars  
 272893 amd64->IR: 0x66 0xF 0x38 0x2B 0xC1 0x66 0xF 0x7F == (closed as dup)  
 272955 Unhandled syscall error for pwrite64 on ppc64 arch  
 272967 make documentation build-system more robust  
 272986 Fix gcc-4.6 warnings with valgrind.h  
 273318 amd64->IR: 0x66 0xF 0x3A 0x61 0xC1 0x38 (missing PCMPxSTRx case)  
 273318 unhandled PCMPxSTRx case: vex amd64->IR: 0x66 0xF 0x3A 0x61 0xC1 0x38  
 273431 valgrind segfaults in evalCfiExpr (debuginfo.c:2039)  
 273465 Callgrind: jumps.c:164 (new\_jcc): Assertion '(0 <= jmp) && ...'  
 273536 Build error: multiple definition of `vgDrd\_pthread\_cond\_initializer'  
 273640 ppc64-linux: unhandled syscalls setresuid(164) and setresgid(169)  
 273729 == 283000 (Illegal opcode for SSE2 "roundsd" instruction)  
 273778 exp-ptrcheck: unhandled sysno == 259  
 274089 exp-ptrcheck: unhandled sysno == 208  
 274378 s390x: Various dispatcher tweaks

274447 WARNING: unhandled syscall: 340  
 274776 amd64->IR: 0x66 0xF 0x38 0x2B 0xC5 0x66  
 274784 == 267997 (valgrind ls -l results in Segmentation Fault)  
 274926 valgrind does not build against linux-3  
 275148 configure FAIL with glibc-2.14  
 275151 Fedora 15 / glibc-2.14 'make regtest' FAIL  
 275168 Make Valgrind work for MacOSX 10.7 Lion  
 275212 == 275284 (lots of false positives from \_\_memcpy\_ssse3\_back et al)  
 275278 valgrind does not build on Linux kernel 3.0.\* due to silly  
 275284 Valgrind memcpy/memmove redirection stopped working in glibc 2.14/x86\_64  
 275308 Fix implementation for ppc64 fres instruc  
 275339 s390x: fix testcase compile warnings  
 275517 s390x: Provide support for CKSM instruction  
 275710 s390x: get rid of redundant address mode calculation  
 275815 == 247894 (Valgrind doesn't know about Linux readahead(2) syscall)  
 275852 == 250101 (valgrind uses all swap space and is killed)  
 276784 Add support for IBM Power ISA 2.06 -- stage 3  
 276987 gdbsrv: fix tests following recent commits  
 277045 Valgrind crashes with unhandled DW\_OP\_opcode 0x2a  
 277199 The test\_isa\_2\_06\_part1.c in none/tests/ppc64 should be a symlink  
 277471 Unhandled syscall: 340  
 277610 valgrind crashes in VG\_(lseek)(core\_fd, phdrs[idx].p\_offset, ...)  
 277653 ARM: support Thumb2 PLD instruction  
 277663 ARM: NEON float VMUL by scalar incorrect  
 277689 ARM: tests for VSTn with register post-index are broken  
 277694 ARM: BLX LR instruction broken in ARM mode  
 277780 ARM: VMOV.F32 (immediate) instruction is broken  
 278057 fuse filesystem syscall deadlocks  
 278078 Unimplemented syscall 280 on ppc32  
 278349 F\_GETPIPE\_SZ and F\_SETPIPE\_SZ Linux fcntl commands  
 278454 VALGRIND\_STACK\_DEREGISTER has wrong output type  
 278502 == 275284 (Valgrind confuses memcpy() and memmove())  
 278892 gdbsrv: factorize gdb version handling, fix doc and typos  
 279027 Support for MVCL and CLCL instruction  
 279027 s390x: Provide support for CLCL and MVCL instructions  
 279062 Remove a redundant check in the insn selector for ppc.  
 279071 JDK creates PTEST with redundant REX.W prefix  
 279212 gdbsrv: add monitor cmd v.info scheduler.  
 279378 exp-ptrcheck: the 'impossible' happened on mkfifo call  
 279698 memcheck discards valid-bits for packuswb  
 279795 memcheck reports uninitialised values for mincore on amd64  
 279994 Add support for IBM Power ISA 2.06 -- stage 3  
 280083 mempolicy syscall check errors  
 280290 vex amd64->IR: 0x66 0xF 0x38 0x28 0xC1 0x66 0xF 0x6F  
 280710 s390x: config files for nightly builds  
 280757 /tmp dir still used by valgrind even if TMPDIR is specified  
 280965 Valgrind breaks fcntl locks when program does mmap  
 281138 WARNING: unhandled syscall: 340  
 281241 == 275168 (valgrind useless on MacOS 10.7.1 Lion)  
 281304 == 275168 (Darwin: dyld "cannot load inserted library")  
 281305 == 275168 (unhandled syscall: unix:357 on Darwin 11.1)  
 281468 s390x: handle do\_clone and gcc clones in call traces  
 281488 ARM: VFP register corruption  
 281828 == 275284 (false memmove warning: "Source and destination overlap")  
 281883 s390x: Fix system call wrapper for "clone".  
 282105 generalise 'reclaimSuperBlock' to also reclaim splittable superblock  
 282112 Unhandled instruction bytes: 0xDE 0xD9 0x9B 0xDF (fcompp)  
 282238 SLES10: make check fails

282979 strcasestr needs replacement with recent(>=2.12) glibc  
 283000 vex amd64->IR: 0x66 0xF 0x3A 0xA 0xC0 0x9 0xF3 0xF  
 283243 Regression in ppc64 memcheck tests  
 283325 == 267997 (Darwin: V segfaults on startup when built with Xcode 4.0)  
 283427 re-connect epoll\_pwait syscall on ARM linux  
 283600 gdbsrv: android: port vgdb.c  
 283709 none/tests/faultstatus needs to account for page size  
 284305 filter\_gdb needs enhancement to work on ppc64  
 284384 clang 3.1 -Wunused-value warnings in valgrind.h, memcheck.h  
 284472 Thumb2 ROR.W encoding T2 not implemented  
 284621 XML-escape process command line in XML output  
 n-i-bz cachegrind/callgrind: handle CPUID information for Core iX Intel CPUs  
       that have non-power-of-2 sizes (also AMDs)  
 n-i-bz don't be spooked by libraries mashed by elfhack  
 n-i-bz don't be spooked by libxul.so linked with gold  
 n-i-bz improved checking for VALGRIND\_CHECK\_MEM\_IS\_DEFINED

(3.7.0-TEST1: 27 October 2011, vex r2228, valgrind r12245)  
 (3.7.0.RC1: 1 November 2011, vex r2231, valgrind r12257)  
 (3.7.0: 5 November 2011, vex r2231, valgrind r12258)

#### Release 3.6.1 (16 February 2011)

~~~~~

3.6.1 is a bug fix release. It adds support for some SSE4 instructions that were omitted in 3.6.0 due to lack of time. Initial support for glibc-2.13 has been added. A number of bugs causing crashing or assertion failures have been fixed.

The following bugs have been fixed or resolved. Note that "n-i-bz" stands for "not in bugzilla" -- that is, a bug that was reported to us but never got a bugzilla entry. We encourage you to file bugs in bugzilla ([http://bugs.kde.org/enter\\_valgrind\\_bug.cgi](http://bugs.kde.org/enter_valgrind_bug.cgi)) rather than mailing the developers (or mailing lists) directly -- bugs that are not entered into bugzilla tend to get forgotten about or ignored.

To see details of a given bug, visit  
[https://bugs.kde.org/show\\_bug.cgi?id=XXXXXX](https://bugs.kde.org/show_bug.cgi?id=XXXXXX)  
 where XXXXXX is the bug number as listed below.

188572 Valgrind on Mac should suppress setenv() mem leak  
 194402 vex amd64->IR: 0x48 0xF 0xAE 0x4 (proper FX{SAVE,RSTOR} support)  
 210481 vex amd64->IR: Assertion `sz == 2 || sz == 4' failed (REX.W POPQ)  
 246152 callgrind internal error after pthread\_cancel on 32 Bit Linux  
 250038 ppc64: Altivec LVSR and LVSL instructions fail their regtest  
 254420 memory pool tracking broken  
 254957 Test code failing to compile due to changes in memcheck.h  
 255009 helgrind/drd: crash on chmod with invalid parameter  
 255130 readdwarf3.c parse\_type\_DIE confused by GNAT Ada types  
 255355 helgrind/drd: crash on threaded programs doing fork  
 255358 == 255355  
 255418 (SSE4.x) rint call compiled with ICC  
 255822 --gen-suppressions can create invalid files: "too many callers [...]"  
 255888 closing valgrindoutput tag outputted to log-stream on error  
 255963 (SSE4.x) vex amd64->IR: 0x66 0xF 0x3A 0x9 0xDB 0x0 (ROUNDPD)  
 255966 Slowness when using mempool annotations  
 256387 vex x86->IR: 0xD4 0xA 0x2 0x7 (AAD and AAM)

256600 super-optimized strcasecmp() false positive  
 256669 vex amd64->IR: Unhandled LOOPNEL insn on amd64  
 256968 (SSE4.x) vex amd64->IR: 0x66 0xF 0x38 0x10 0xD3 0x66 (BLENDVPx)  
 257011 (SSE4.x) vex amd64->IR: 0x66 0xF 0x3A 0xE 0xFD 0xA0 (PBLENDW)  
 257063 (SSE4.x) vex amd64->IR: 0x66 0xF 0x3A 0x8 0xC0 0x0 (ROUNDPS)  
 257276 Missing case in memcheck --track-origins=yes  
 258870 (SSE4.x) Add support for EXTRACTPS SSE 4.1 instruction  
 261966 (SSE4.x) support for CRC32B and CRC32Q is lacking (also CRC32{W,L})  
 262985 VEX regression in valgrind 3.6.0 in handling PowerPC VMX  
 262995 (SSE4.x) crash when trying to valgrind gcc-snapshot (PCMPxSTRx \$0)  
 263099 callgrind\_annotate counts Ir improperly [...]  
 263877 undefined coprocessor instruction on ARMv7  
 265964 configure FAIL with glibc-2.13  
 n-i-bz Fix compile error w/ icc-12.x in guest\_arm\_toIR.c  
 n-i-bz Docs: fix bogus descriptions for VALGRIND\_CREATE\_BLOCK et al  
 n-i-bz Massif: don't assert on shmat() with --pages-as-heap=yes  
 n-i-bz Bug fixes and major speedups for the exp-DHAT space profiler  
 n-i-bz DRD: disable --free-is-write due to implementation difficulties

(3.6.1: 16 February 2011, vex r2103, valgrind r11561).

# 4. README

Release notes for Valgrind

~~~~~

If you are building a binary package of Valgrind for distribution, please read README\_PACKAGERS. It contains some important information.

If you are developing Valgrind, please read README\_DEVELOPERS. It contains some useful information.

For instructions on how to build/install, see the end of this file.

If you have problems, consult the FAQ to see if there are workarounds.

Executive Summary

~~~~~

Valgrind is a framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes seven production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and two heap profilers. It also includes one experimental tool: a SimPoint basic block vector generator.

Valgrind is closely tied to details of the CPU, operating system and to a lesser extent, compiler and basic C libraries. This makes it difficult to make it portable. Nonetheless, it is available for the following platforms:

- X86/Linux
- AMD64/Linux
- PPC32/Linux
- PPC64/Linux
- ARM/Linux
- ARM64/Linux
- x86/macOS
- AMD64/macOS
- S390X/Linux
- MIPS32/Linux
- MIPS64/Linux
- nanoMIPS/Linux
- X86/Solaris
- AMD64/Solaris
- X86/FreeBSD
- AMD64/FreeBSD

Note that AMD64 is just another name for x86\_64, and Valgrind runs fine on Intel processors. Also note that the core of macOS is called "Darwin" and this name is used sometimes.

Valgrind is licensed under the GNU General Public License, version 2. Read the file COPYING in the source distribution for details.

However: if you contribute code, you need to make it available as GPL version 2 or later, and not 2-only.

## Documentation

~~~~~

A comprehensive user guide is supplied. Point your browser at \$PREFIX/share/doc/valgrind/manual.html, where \$PREFIX is whatever you specified with --prefix= when building.

## Building and installing it

~~~~~

To install from the GIT repository:

0. Clone the code from GIT:  
`git clone https://sourceware.org/git/valgrind.git`  
 There are further instructions at  
<http://www.valgrind.org/downloads/repository.html>.

1. `cd` into the source directory.
2. Run `./autogen.sh` to setup the environment (you need the standard autoconf tools to do so).
3. Continue with the following instructions...

To install from a tar.bz2 distribution:

4. Run `./configure`, with some options if you wish. The only interesting one is the usual `--prefix=/where/you/want/it/installed`.
5. Run `"make"`.
6. Run `"make install"`, possibly as root if the destination permissions require that.
7. See if it works. Try `"valgrind ls -l"`. Either this works, or it bombs out with some complaint. In that case, please let us know (see [http://valgrind.org/support/bug\\_reports.html](http://valgrind.org/support/bug_reports.html)).

Important! Do not move the valgrind installation into a place different from that specified by --prefix at build time. This will cause things to break in subtle ways, mostly when Valgrind handles fork/exec calls.

The Valgrind Developers

# 5. README\_MISSING\_SYSCALL\_OR\_IOCTL

Dealing with missing system call or ioctl wrappers in Valgrind

~~~~~

You're probably reading this because Valgrind bombed out whilst running your program, and advised you to read this file. The good news is that, in general, it's easy to write the missing syscall or ioctl wrappers you need, so that you can continue your debugging. If you send the resulting patches to me, then you'll be doing a favour to all future Valgrind users too.

Note that an "ioctl" is just a special kind of system call, really; so there's not a lot of need to distinguish them (at least conceptually) in the discussion that follows.

All this machinery is in coregrind/m\_syswrap.

What are syscall/ioctl wrappers? What do they do?

~~~~~

Valgrind does what it does, in part, by keeping track of everything your program does. When a system call happens, for example a request to read part of a file, control passes to the Linux kernel, which fulfils the request, and returns control to your program. The problem is that the kernel will often change the status of some part of your program's memory as a result, and tools (instrumentation plug-ins) may need to know about this.

Syscall and ioctl wrappers have two jobs:

1. Tell a tool what's about to happen, before the syscall takes place. A tool could perform checks beforehand, eg. if memory about to be written is actually writable. This part is useful, but not strictly essential.
2. Tell a tool what just happened, after a syscall takes place. This is so it can update its view of the program's state, eg. that memory has just been written to. This step is essential.

The "happenings" mostly involve reading/writing of memory.

So, let's look at an example of a wrapper for a system call which should be familiar to many Unix programmers.

The syscall wrapper for time()

~~~~~

The wrapper for the time system call looks like this:

```
PRE(sys_time)
{
    /* time_t time(time_t *t); */
    PRINT("sys_time ( %p )",ARG1);
    PRE_REG_READ1(long, "time", int *, t);
    if (ARG1 != 0) {
```

```
    PRE_MEM_WRITE( "time(t)", ARG1, sizeof(vki_time_t) );
}
}

POST(sys_time)
{
    if (ARG1 != 0) {
        POST_MEM_WRITE( ARG1, sizeof(vki_time_t) );
    }
}
```

The first thing we do happens before the syscall occurs, in the PRE() function. The PRE() function typically starts with invoking to the PRINT() macro. This PRINT() macro implements support for the --trace-syscalls command line option. Next, the tool is told the return type of the syscall, that the syscall has one argument, the type of the syscall argument and that the argument is being read from a register:

```
PRE_REG_READ1(long, "time", int *, t);
```

Next, if a non-NULL buffer is passed in as the argument, tell the tool that the buffer is about to be written to:

```
if (ARG1 != 0) {
    PRE_MEM_WRITE( "time", ARG1, sizeof(vki_time_t) );
}
```

Finally, the really important bit, after the syscall occurs, in the POST() function: if, and only if, the system call was successful, tell the tool that the memory was written:

```
if (ARG1 != 0) {
    POST_MEM_WRITE( ARG1, sizeof(vki_time_t) );
}
```

The POST() function won't be called if the syscall failed, so you don't need to worry about checking that in the POST() function. (Note: this is sometimes a bug; some syscalls do return results when they "fail" - for example, nanosleep returns the amount of unslept time if interrupted. TODO: add another per-syscall flag for this case.)

Note that we use the type 'vki\_time\_t'. This is a copy of the kernel type, with 'vki\_' prefixed. Our copies of such types are kept in the appropriate vki\*.h file(s). We don't include kernel headers or glibc headers directly.

Writing your own syscall wrappers (see below for ioctl wrappers)

~~~~~  
If Valgrind tells you that system call NNN is unimplemented, do the following:

1. Find out the name of the system call:

```
grep NNN /usr/include/asm/unistd*.h
```

This should tell you something like \_\_NR\_mysyscallname.

Copy this entry to include/vki/vki-scnums-\$(VG\_PLATFORM).h.

If you can't find the system call in /usr/include, try looking in the strace source code (<https://github.com/strace/strace>). Some syscalls/ioctls are not defined explicitly, but strace may have already figured it out.

2. Do 'man 2 mysyscallname' to get some idea of what the syscall does. Note that the actual kernel interface can differ from this, so you might also want to check a version of the Linux kernel source.

NOTE: any syscall which has something to do with signals or threads is probably "special", and needs more careful handling. Post something to valgrind-developers if you aren't sure.

3. Add a case to the already-huge collection of wrappers in the coregrind/m\_syswrap/syswrap-\*.c files.  
For each in-memory parameter which is read or written by the syscall, do one of

```
PRE_MEM_READ( ... )  
PRE_MEM_RASCIIZ( ... )  
PRE_MEM_WRITE( ... )
```

for that parameter. Then do the syscall. Then, if the syscall succeeds, issue suitable POST\_MEM\_WRITE( ... ) calls.  
(There's no need for POST\_MEM\_READ calls.)

Also, add it to the syscall\_table[] array; use one of GENX\_, GENXY, LINX\_, LINXY, PLAX\_, PLAXY.

GEN\* for generic syscalls (in syswrap-generic.c), LIN\* for linux specific ones (in syswrap-linux.c) and PLA\* for the platform dependent ones (in syswrap-\$(PLATFORM)-linux.c).

The \*XY variant if it requires a PRE() and POST() function, and the \*X\_ variant if it only requires a PRE() function.

If you find this difficult, read the wrappers for other syscalls for ideas. A good tip is to look for the wrapper for a syscall which has a similar behaviour to yours, and use it as a starting point.

If you need structure definitions and/or constants for your syscall, copy them from the kernel headers into include/vki.h and co., with the appropriate vki\_\*/VKI\_\* name mangling. Don't #include any kernel headers. And certainly don't #include any glibc headers.

Test it.

Note that a common error is to call POST\_MEM\_WRITE( ... ) with 0 (NULL) as the first (address) argument. This usually means your logic is slightly inadequate. It's a sufficiently common bug that there's a built-in check for it, and you'll get a "probably sanity check failure" for the syscall wrapper you just made, if this is the case.

4. Once happy, send us the patch. Pretty please.

#### Writing your own ioctl wrappers

~~~~~

Is pretty much the same as writing syscall wrappers, except that all the action happens within PRE(ioctl) and POST(ioctl).

There's a default case, sometimes it isn't correct and you have to write a more specific case to get the right behaviour.

As above, please create a bug report and attach the patch as described on <http://www.valgrind.org>.

#### Writing your own door call wrappers (Solaris only)

~~~~~

Unlike syscalls or ioctls, door calls transfer data between two userspace programs, albeit through a kernel interface. Programs may use completely proprietary semantics in the data buffers passed between them. Therefore it may not be possible to capture these semantics within a Valgrind door call or door return wrapper.

Nevertheless, for system or well-known door services it would be beneficial to have a door call and a door return wrapper. Writing such wrapper is pretty much the same as writing ioctl wrappers. Please take a few moments to study the following picture depicting how a door client and a door server interact through the kernel interface in a typical scenario:

```

door client thread      kernel      door server thread
invokes door_call()      invokes door_return()
-----
                <---- PRE(sys_door, DOOR_RETURN)
PRE(sys_door, DOOR_CALL) --->
                ----> POST(sys_door, DOOR_RETURN)
                        ----> server_procedure()
                        <----
                <---- PRE(sys_door, DOOR_RETURN)
POST(sys_door, DOOR_CALL) <---
```

The first PRE(sys\_door, DOOR\_RETURN) is invoked with data\_ptr=NULL and data\_size=0. That's because it has not received any data from a door call, yet.

Semantics are described by the following functions in coregring/m\_syswrap/syswrap-solaris.c module:

- o For a door call wrapper the following attributes of 'params' argument:
  - data\_ptr (and associated data\_size) as input buffer (request);
    - described in door\_call\_pre\_mem\_params\_data()
  - rbuf (and associated rsize) as output buffer (response);
    - described in door\_call\_post\_mem\_params\_rbuf()
- o For a door return wrapper the following parameters:

- data\_ptr (and associated data\_size) as input buffer (request);  
described in door\_return\_post\_mem\_data()
- data\_ptr (and associated data\_size) as output buffer (response);  
described in door\_return\_pre\_mem\_data()

There's a default case which may not be correct and you have to write a more specific case to get the right behaviour. Unless Valgrind's option '--sim-hints=lax-doors' is specified, the default case also spits a warning.

As above, please create a bug report and attach the patch as described on <http://www.valgrind.org>.

## 6. README\_DEVELOPERS

### Building and installing it

~~~~~

To build/install from the GIT repository or from a distribution tarball, refer to the section with the same name in README.

### Building and not installing it

~~~~~

To run Valgrind without having to install it, run `coregrind/valgrind` with the `VALGRIND_LIB` environment variable set, where `<dir>` is the root of the source tree (and must be an absolute path). Eg:

```
VALGRIND_LIB=~/.grind/head4/.in_place ~/.grind/head4/coregrind/valgrind
```

This allows you to compile and run with "make" instead of "make install", saving you time.

Or, you can use the 'vg-in-place' script which does that for you.

I recommend compiling with "make --quiet" to further reduce the amount of output spewed out during compilation, letting you actually see any errors, warnings, etc.

### Building a distribution tarball

~~~~~

To build a distribution tarball from the valgrind sources:

```
make dist
```

In addition to compiling, linking and packaging everything up, the command will also attempt to build the documentation.

If you only want to test whether the generated tarball is complete and runs regression tests successfully, building documentation is not needed.

```
make dist BUILD_ALL_DOCS=no
```

If you insist on building documentation some embarrassing instructions can be found in docs/README.

### Running the regression tests

~~~~~

To build and run all the regression tests, run "make [--quiet] regtest".

To run a subset of the regression tests, execute:

```
perl tests/vg_regtest <name>
```

where `<name>` is a directory (all tests within will be run) or a single `.vgtest` test file, or the name of a program which has a like-named `.vgtest` file. Eg:

```
perl tests/vg_regtest memcheck
perl tests/vg_regtest memcheck/tests/badfree.vgtest
perl tests/vg_regtest memcheck/tests/badfree
```

### Running the performance tests

~~~~~

To build and run all the performance tests, run "make [--quiet] perf".

To run a subset of the performance suite, execute:

```
perl perf/vg_perf <name>
```

where <name> is a directory (all tests within will be run) or a single .vgperf test file, or the name of a program which has a like-named .vgperf file. Eg:

```
perl perf/vg_perf perf/
perl perf/vg_perf perf/bz2.vgperf
perl perf/vg_perf perf/bz2
```

To compare multiple versions of Valgrind, use the --vg= option multiple times. For example, if you have two Valgrinds next to each other, one in trunk1/ and one in trunk2/, from within either trunk1/ or trunk2/ do this to compare them on all the performance tests:

```
perl perf/vg_perf --vg=../trunk1 --vg=../trunk2 perf/
```

### Commit access and try branches

~~~~~

To get commit access to the valgrind git repository on sourceware you will have to ask an existing developer and fill in the following form: [https://sourceware.org/cgi-bin/pdw/ps\\_form.cgi](https://sourceware.org/cgi-bin/pdw/ps_form.cgi)

Every developer with commit access can use try branches. If you want to try a branch before pushing you can push to a special named try branch as follows:

```
git push origin $BRANCH:users/$USERNAME/try-$BRANCH
```

Where \$BRANCH is the branch name and \$USERNAME is your user name.

You can see the status of the builders here:

<https://builder.sourceware.org/buildbot/#/builders?tags=valgrind-try>

The buildbot will also sent the patch author multiple success/failure emails.

Afterwards you can delete the branch again:

```
git push origin :users/$USERNAME/try-$BRANCH
```

### Debugging Valgrind with GDB

~~~~~

To debug the valgrind launcher program (<prefix>/bin/valgrind) just run it under gdb in the normal way.

Debugging the main body of the valgrind code (and/or the code for

a particular tool) requires a bit more trickery but can be achieved without too much problem by following these steps:

- (1) Set VALGRIND\_LAUNCHER to point to the valgrind executable. Eg:

```
export VALGRIND_LAUNCHER=/usr/local/bin/valgrind
```

or for an uninstalled version in a source directory \$DIR:

```
export VALGRIND_LAUNCHER=$DIR/coregrind/valgrind
export VALGRIND_LIB=$DIR/.in_place
```

VALGRIND\_LIB is where the default.supp and vgpreload\_ libraries are found (which is under /usr/libexec/valgrind for an installed version).

- (2) Run gdb on the tool executable. Eg:

```
gdb /usr/local/lib/valgrind/lackey-ppc32-linux
```

or

```
gdb $DIR/.in_place/memcheck-x86-linux
```

- (3) Do "handle SIGSEGV SIGILL nostop noprint" in GDB to prevent GDB from stopping on a SIGSEGV or SIGILL:

```
(gdb) handle SIGILL SIGSEGV nostop noprint
```

If you are using lldb, then the equivalent command is

```
(lldb) pro hand -p true -s false -n false SIGILL SIGSEGV
```

- (4) Set any breakpoints you want and proceed as normal for gdb. The macro VG\_(FUNC) is expanded to vgPlain\_FUNC, so If you want to set a breakpoint VG\_(do\_exec), you could do like this in GDB:

```
(gdb) b vgPlain_do_exec
```

- (5) Run the tool with required options (the --tool option is required for correct setup), e.g.

```
(gdb) run --tool=lackey pwd
```

Steps (1)--(3) can be put in a .gdbinit file, but any directory names must be fully expanded (ie. not an environment variable).

A different and possibly easier way is as follows:

- (1) Run Valgrind as normal, but add the flag --wait-for-gdb=yes. This puts the tool executable into a wait loop soon after it gains control. This delays startup for a few seconds.

- (2) In a different shell, do "gdb /proc/<pid>/exe <pid>", where <pid> you read from the output printed by (1). This attaches GDB to the tool executable, which should be in the above mentioned wait loop.

- (3) Do "cont" to continue. After the loop finishes spinning, startup will continue as normal. Note that comment (3) above re passing signals applies here too.

The default build of Valgrind uses "-g -O2". This is OK most of the time, but with sophisticated optimization it can be difficult to see the contents of variables. A quick way to get to see function variables is to temporarily add "\_\_attribute\_\_((optnone))" before the function definition and rebuild. Alternatively modify Makefile.all.am and remove -O2 from AM\_CFLAGS\_BASE. That will require you to reconfigure and rebuild Valgrind.

### Self-hosting

~~~~~

This section explains:

- (A) How to configure Valgrind to run under Valgrind.  
Such a setup is called self hosting, or outer/inner setup.
- (B) How to run Valgrind regression tests in a 'self-hosting' mode, e.g. to verify Valgrind has no bugs such as memory leaks.
- (C) How to run Valgrind performance tests in a 'self-hosting' mode, to analyse and optimise the performance of Valgrind and its tools.

(A) How to configure Valgrind to run under Valgrind:

- (1) Check out 2 trees, "Inner" and "Outer". Inner runs the app directly. Outer runs Inner.
- (2) Configure Inner with --enable-inner and build as usual.
- (3) Configure Outer normally and build+install as usual.  
Note: You must use a "make install"-ed valgrind.  
Do *\*not\** use vg-in-place for the Outer valgrind.

(4) Choose a very simple program (date) and try

```
outer/.../bin/valgrind --sim-hints=enable-outer --trace-children=yes \  
  --smc-check=all-non-file \  
  --run-libc-freeres=no --tool=cachegrind -v \  
inner/.../vg-in-place --vgdb-prefix=./inner --tool=none -v prog
```

If you omit the --trace-children=yes, you'll only monitor Inner's launcher program, not its stage2. Outer needs --run-libc-freeres=no, as otherwise it will try to find and run \_\_libc\_freeres in the inner, while libc is not used by the inner. Inner needs --vgdb-prefix=./inner to avoid inner gdbserver colliding with outer gdbserver.

Currently, inner does *\*not\** use the client request

VALGRIND\_DISCARD\_TRANSLATIONS for the JITted code or the code patched for translation chaining. So the outer needs --smc-check=all-non-file to detect the modified code.

Debugging the whole thing might imply to use up to 3 GDB:

- \* a GDB attached to the Outer valgrind, allowing to examine the state of Outer.
- \* a GDB using Outer gdbserver, allowing to examine the state of Inner.
- \* a GDB using Inner gdbserver, allowing to examine the state of prog.

The whole thing is fragile, confusing and slow, but it does work well enough for you to get some useful performance data. Inner has most of its output (ie. those lines beginning with "`==<pid>==`") prefixed with a '>', which helps a lot. However, when running regression tests in an Outer/Inner setup, this prefix causes the reg test diff to fail. Give `--sim-hints=no-inner-prefix` to the Inner to disable the production of the prefix in the stdout/stderr output of Inner.

The allocators in `coregrind/m_mallocfree.c` and `VEX/priv/main_util.h` are annotated with client requests so Memcheck can be used to find leaks and use after free in an Inner Valgrind.

The Valgrind "big lock" is annotated with helgrind client requests so Helgrind and DRD can be used to find race conditions in an Inner Valgrind.

All this has not been tested much, so don't be surprised if you hit problems.

When using self-hosting with an outer Callgrind tool, use '`--pop-on-jump`' (on the outer). Otherwise, Callgrind has much higher memory requirements.

(B) Regression tests in an outer/inner setup:

To run all the regression tests with an outer memcheck, do :

```
perl tests/vg_regtest --outer-valgrind=./outer/.../bin/valgrind \
--all
```

To run a specific regression tests with an outer memcheck, do:

```
perl tests/vg_regtest --outer-valgrind=./outer/.../bin/valgrind \
none/tests/args.vgtest
```

To run regression tests with another outer tool:

```
perl tests/vg_regtest --outer-valgrind=./outer/.../bin/valgrind \
--outer-tool=helgrind --all
```

`--outer-args` allows to give specific arguments to the outer tool, replacing the default one provided by `vg_regtest`.

Note: `--outer-valgrind` must be a "make install"-ed valgrind.  
Do *\*not\** use `vg-in-place`.

When an outer valgrind runs an inner valgrind, a regression test produces one additional file `<testname>.outer.log` which contains the errors detected by the outer valgrind. E.g. for an outer memcheck, it contains the leaks found in the inner, for an outer helgrind or drd, it contains the detected race conditions.

The file `tests/outer_inner.supp` contains suppressions for the irrelevant or benign errors found in the inner.

A regression test running in the inner (e.g. `memcheck/tests/badrw`) will cause the inner to report an error, which is expected and checked as usual when running the regtests in an outer/inner setup. However, the outer will often also observe an error, e.g. a jump using uninitialised data, or a read/write outside the bounds of a heap block. When the outer reports such an error, it will output the inner host stacktrace. To this stacktrace, it will append the stacktrace of the inner guest program. For example, this is an error

reported by the outer when the inner runs the badrw regtest:

```

==8119== Invalid read of size 2
==8119==   at 0x7F2EFD7AF: ???
==8119==   by 0x7F2C82EAF: ???
==8119==   by 0x7F180867F: ???
==8119==   by 0x40051D: main (badrw.c:5)
==8119==   by 0x7F180867F: ???
==8119==   by 0x1BFF: ???
==8119==   by 0x3803B7F0: _____VVVVVVVVV_appended_inner_guest_stack_VVVVVVVV_____ (m_execontext.c:332)
==8119==   by 0x40055C: main (badrw.c:22)
==8119== Address 0x55cd03c is 4 bytes before a block of size 16 alloc'd
==8119==   at 0x2804E26D: vgPlain_arena_malloc (m_mallocfree.c:1914)
==8119==   by 0x2800BAB4: vgMemCheck_new_block (mc_malloc_wrappers.c:368)
==8119==   by 0x2800BC87: vgMemCheck_malloc (mc_malloc_wrappers.c:403)
==8119==   by 0x28097EAE: do_client_request (scheduler.c:1861)
==8119==   by 0x28097EAE: vgPlain_scheduler (scheduler.c:1425)
==8119==   by 0x280A7237: thread_wrapper (syswrap-linux.c:103)
==8119==   by 0x280A7237: run_a_thread_NORETURN (syswrap-linux.c:156)
==8119==   by 0x3803B7F0: _____VVVVVVVVV_appended_inner_guest_stack_VVVVVVVV_____ (m_execontext.c:332)
==8119==   by 0x4C294C4: malloc (vg_replace_malloc.c:298)
==8119==   by 0x40051D: main (badrw.c:5)

```

In the above, the first stacktrace starts with the inner host stacktrace, which in this case is some JITted code. Such code sometimes contains IPs that points in the inner guest code (0x40051D: main (badrw.c:5)).

After the separator, we have the inner guest stacktrace.

The second stacktrace gives the stacktrace where the heap block that was overrun was allocated. We see it was allocated by the inner valgrind in the client arena (first part of the stacktrace). The second part is the guest stacktrace that did the allocation.

### (C) Performance tests in an outer/inner setup:

To run all the performance tests with an outer cachegrind, do :

```
perl perf/vg_perf --outer-valgrind=./outer/.../bin/valgrind perf
```

To run a specific perf test (e.g. bz2) in this setup, do :

```
perl perf/vg_perf --outer-valgrind=./outer/.../bin/valgrind perf/bz2
```

To run all the performance tests with an outer callgrind, do :

```
perl perf/vg_perf --outer-valgrind=./outer/.../bin/valgrind \
--outer-tool=callgrind perf
```

Note: --outer-valgrind must be a "make install"-ed valgrind.

Do *\*not\** use vg-in-place.

To compare the performance of multiple Valgrind versions, do :

```
perl perf/vg_perf --outer-valgrind=./outer/.../bin/valgrind \
--outer-tool=callgrind \
--vg=./inner_xxxx --vg=./inner_yyyy perf
```

(where inner\_xxxx and inner\_yyyy are the toplevel directories of the versions to compare).

Cachegrind and cg\_diff are particularly handy to obtain a delta between the two versions.

When the outer tool is callgrind or cachegrind, the following output files will be created for each test:

```
<outertoolname>.out.<inner_valgrind_dir>.<tt>.<perftestname>.<pid>
```

<outertoolname>.outer.log.<inner\_valgrind\_dir>.<tt>.<perftestname>.<pid>  
(where tt is the two letters abbreviation for the inner tool(s) run).

For example, the command

```
perl perf/vg_perf \  
  --outer-valgrind=./outer_trunk/install/bin/valgrind \  
  --outer-tool=callgrind \  
  --vg=./inner_tchain --vg=./inner_trunk perf/many-loss-records
```

produces the files

```
callgrind.out.inner_tchain.no.many-loss-records.18465  
callgrind.out.log.inner_tchain.no.many-loss-records.18465  
callgrind.out.inner_tchain.me.many-loss-records.21899  
callgrind.out.log.inner_tchain.me.many-loss-records.21899  
callgrind.out.inner_trunk.no.many-loss-records.21224  
callgrind.out.log.inner_trunk.no.many-loss-records.21224  
callgrind.out.inner_trunk.me.many-loss-records.22916  
callgrind.out.log.inner_trunk.me.many-loss-records.22916
```

Printing out problematic blocks

~~~~~

If you want to print out a disassembly of a particular block that causes a crash, do the following.

Try running with "--vex-guest-chase=no --trace-flags=10000000 --trace-notbelow=999999". This should print one line for each block translated, and that includes the address.

Then re-run with 999999 changed to the highest bb number shown.

This will print the one line per block, and also will print a disassembly of the block in which the fault occurred.

Formatting the code with clang-format

~~~~~

clang-format is a tool to format C/C++/... code. The root directory of the Valgrind tree contains file .clang-format which is a configuration for this tool and specifies a style for Valgrind. This gives you an option to use clang-format to easily format Valgrind code which you are modifying.

The Valgrind codebase is not globally formatted with clang-format. It means that you should not use the tool to format a complete file after making changes in it because that would lead to creating unrelated modifications.

The right approach is to format only updated or new code. By using an integration with a text editor, it is possible to reformat arbitrary blocks of code with a single keystroke. Refer to the upstream documentation which describes integration with various editors and IDEs:  
<https://clang.llvm.org/docs/ClangFormat.html>.

# 7. README\_PACKAGERS

Greetings, packaging person! This information is aimed at people building binary distributions of Valgrind.

Thanks for taking the time and effort to make a binary distribution of Valgrind. The following notes may save you some trouble.

-- If your toolchain (compiler, linker) support lto, using the configure option `--enable-lto=yes` will produce a smaller/faster valgrind (up to 10%).

-- Do not ship your Linux distro with a completely stripped `/lib/ld.so`. At least leave the debugging symbol names on -- line number info isn't necessary. If you don't want to leave symbols on `ld.so`, alternatively you can have your distro install `ld.so's debuginfo` package by default, or make `ld.so.debuginfo` be a requirement of your Valgrind RPM/DEB/whatever.

Reason for this is that Valgrind's Memcheck tool needs to intercept calls to, and provide replacements for, some symbols in `ld.so` at startup (most importantly `strlen`). If it cannot do that, Memcheck shows a large number of false positives due to the highly optimised `strlen` (etc) routines in `ld.so`. This has caused some trouble in the past. As of version 3.3.0, on some targets (ppc32-linux, ppc64-linux), Memcheck will simply stop at startup (and print an error message) if such symbols are not present, because it is infeasible to continue.

It's not like this is going to cost you much space. We only need the symbols for `ld.so` (a few K at most). Not the debug info and not any `debuginfo` or extra symbols for any other libraries.

-- (Unfortunate but true) When you configure to build with the `--prefix=/foo/bar/xyzzy` option, the prefix `/foo/bar/xyzzy` gets baked into valgrind. The consequence is that you must install valgrind at the location specified in the prefix. If you don't, it may appear to work, but will break doing some obscure things, particularly doing `fork()` and `exec()`.

So you can't build a relocatable RPM / whatever from Valgrind.

-- Don't strip the debug info off `lib/valgrind/$platform/vgpreload*.so` in the installation tree. Either Valgrind won't work at all, or it will still work if you do, but will generate less helpful error messages. Here's an example:

```
Mismatched free() / delete / delete []
at 0x40043249: free (vg_clientfuncs.c:171)
by 0x4102BB4E: QGArray::~~QGArray(void) (tools/qgarray.cpp:149)
by 0x4C261C41: PptDoc::~~PptDoc(void) (include/qmemarray.h:60)
by 0x4C261F0E: PptXml::~~PptXml(void) (pptxml.cc:44)
Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
```

```

at 0x4004318C: __builtin_vec_new (vg_clientfuncs.c:152)
by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)

```

This tells you that some memory allocated with `new[]` was freed with `free()`.

Mismatched `free()` / `delete` / `delete []`

```

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

```

This isn't so helpful. Although you can tell there is a mismatch, the names of the allocating and deallocating functions are no longer visible. The same kind of thing occurs in various other messages from `valgrind`.

- Don't strip symbols from `libexec/valgrind/*` in the installation tree. Doing so will likely cause problems. Removing the line number info is probably OK (at least for some of the files in that directory), although that has not been tested by the Valgrind developers.

One consequence of stripping these binaries is that if Valgrind crashes it won't be able to print out a useful callstack. Here is an example posted on Stack Overflow

valgrind: the 'impossible' happened: Killed by fatal signal

host stacktrace:

```

==7732== at 0x38091C12: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x38050E84: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x380510A9: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x380D4F7B: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)
==7732== by 0x380E3946: ??? (in /usr/lib/valgrind/memcheck-amd64-linux)

```

Bug reports like this are less likely to be resolved.

- Please test the final installation works by running it on something huge. I suggest checking that it can start and exit successfully both Firefox and OpenOffice.org. I use these as test programs, and I know they fairly thoroughly exercise Valgrind. The command lines to use are:

```
valgrind -v --trace-children=yes firefox
```

```
valgrind -v --trace-children=yes soffice
```

If you find any more hints/tips for packaging, please report it as a bugreport. See <http://www.valgrind.org> for details.



# 8. README.S390

## Requirements

-----

- You need GCC 3.4 or later to compile the s390 port.
- To run valgrind a z10 machine or any later model is recommended. Older machine models down to and including z990 may work but have not been tested extensively.

## Limitations

-----

- 31-bit client programs are not supported.
- Hexadecimal floating point is not supported.
- Transactional memory is not supported. The transactional-execution facility is masked off from HWCAP.
- A full list of unimplemented instructions can be retrieved from ``docs/internals/s390-opcodes.csv'`, by grepping for "not implemented".
- FP signalling is not accurate. E.g., the "compare and signal" instructions behave like their non-signalling counterparts.
- On machine models predating z10, cachegrind will assume a z10 cache architecture. Otherwise, cachegrind will query the hosts cache system and use those parameters.
- Some gcc versions use mvc to copy 4/8 byte values. This will affect certain debug messages. For example, memcheck will complain about 4 one-byte reads/writes instead of just a single read/write.

## Hardware facilities

-----

Valgrind does not require that the host machine has the same hardware facilities as the machine for which the client program was compiled. This is convenient. If possible, the JIT compiler will translate the client instructions according to the facilities available on the host. This means, though, that probing for hardware facilities by issuing instructions from that facility and observing whether SIGILL is thrown may not work. As a consequence, programs that attempt to do so may behave differently. It is believed that this is a rare use case.

## Reading Material

-----

- (1) ELF ABI s390x Supplement  
<https://github.com/IBM/s390x-abi/releases>
- (2) z/Architecture Principles of Operation  
<https://www.ibm.com/support/pages/zarchitecture-principles-operation>
- (3) z/Architecture Reference Summary  
<https://www.ibm.com/support/pages/zarchitecture-reference-summary>

## 9. README.android

How to cross-compile and run on Android. Please read to the end, since there are important details further down regarding crash avoidance and GPU support.

These notes were last updated on 4 Nov 2014, for Valgrind SVN revision 14689/2987.

These instructions are known to work, or have worked at some time in the past, for:

arm:

- Android 4.0.3 running on a (rooted, AOSP build) Nexus S.
- Android 4.0.3 running on Motorola Xoom.
- Android 4.0.3 running on android arm emulator.
- Android 4.1 running on android emulator.
- Android 2.3.4 on Nexus S worked at some time in the past.

x86:

- Android 4.0.3 running on android x86 emulator.

mips32:

- Android 4.1.2 running on android mips emulator.
- Android 4.2.2 running on android mips emulator.
- Android 4.3 running on android mips emulator.
- Android 4.0.4 running on BROADCOM bcm7425

arm64:

- Android 4.5 (?) running on ARM Juno

On android-arm, GDBserver might insert breaks at wrong addresses. Feedback on this welcome.

Other configurations and toolchains might work, but haven't been tested. Feedback is welcome.

Toolchain:

For arm32, x86 and mips32 you need the android-ndk-r6 native development kit. r6b and r7 give a non-completely-working build; see <http://code.google.com/p/android/issues/detail?id=23203>  
For the android emulator, the versions needed and how to install them are described in README.android\_emulator.

You can get android-ndk-r6 from  
<http://dl.google.com/android/ndk/android-ndk-r6-linux-x86.tar.bz2>

For arm64 (aarch64) you need the android-ndk-r10c NDK, from  
[http://dl.google.com/android/ndk/android-ndk-r10c-linux-x86\\_64.bin](http://dl.google.com/android/ndk/android-ndk-r10c-linux-x86_64.bin)

Install the NDK somewhere. Doesn't matter where. Then:

# Modify this (obviously). Note, this "export" command is only done

---

```

# so as to reduce the amount of typing required. None of the commands
# below read it as part of their operation.
#
export NDKROOT=/path/to/android-ndk-r<version>

# Then cd to the root of your Valgrind source tree.
#
cd /path/to/valgrind/source/tree

# After this point, you don't need to modify anything. Just copy and
# paste the commands below.

# Set up toolchain paths.
#
# For ARM
export AR=$NDKROOT/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-ar
export LD=$NDKROOT/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-ld
export CC=$NDKROOT/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-gcc

# For x86
export AR=$NDKROOT/toolchains/x86-4.4.3/prebuilt/linux-x86/bin/i686-android-linux-ar
export LD=$NDKROOT/toolchains/x86-4.4.3/prebuilt/linux-x86/bin/i686-android-linux-ld
export CC=$NDKROOT/toolchains/x86-4.4.3/prebuilt/linux-x86/bin/i686-android-linux-gcc

# For MIPS32
export AR=$NDKROOT/toolchains/mipsel-linux-android-4.8/prebuilt/linux-x86_64/bin/mipsel-linux-android-ar
export LD=$NDKROOT/toolchains/mipsel-linux-android-4.8/prebuilt/linux-x86_64/bin/mipsel-linux-android-ld
export CC=$NDKROOT/toolchains/mipsel-linux-android-4.8/prebuilt/linux-x86_64/bin/mipsel-linux-android-gcc

# For ARM64 (AArch64)
export AR=$NDKROOT/toolchains/aarch64-linux-android-4.9/prebuilt/linux-x86_64/bin/aarch64-linux-android-ar
export LD=$NDKROOT/toolchains/aarch64-linux-android-4.9/prebuilt/linux-x86_64/bin/aarch64-linux-android-ld
export CC=$NDKROOT/toolchains/aarch64-linux-android-4.9/prebuilt/linux-x86_64/bin/aarch64-linux-android-gcc

# Do configuration stuff. Don't mess with the --prefix in the
# configure command below, even if you think it's wrong.
# You may need to set the --with-tmpdir path to something
# different if /sdcard doesn't work on the device -- this is
# a known cause of difficulties.

# The below re-generates configure, Makefiles, ...
# This is not needed if you start from a release tarball.
./autogen.sh

# for ARM
CPPFLAGS="--sysroot=$NDKROOT/platforms/android-3/arch-arm" \
CFLAGS="--sysroot=$NDKROOT/platforms/android-3/arch-arm" \
./configure --prefix=/data/local/Inst \
--host=armv7-unknown-linux --target=armv7-unknown-linux \
--with-tmpdir=/sdcard
# note: on android emulator, android-14 platform was also tested and works.
# It is not clear what this platform nr really is.

# for x86

```

---

```

CPPFLAGS="--sysroot=$NDKROOT/platforms/android-9/arch-x86" \
CFLAGS="--sysroot=$NDKROOT/platforms/android-9/arch-x86 -fno-pic" \
./configure --prefix=/data/local/Inst \
--host=i686-android-linux --target=i686-android-linux \
--with-tmpdir=/sdcard

```

```
# for MIPS32
```

```

CPPFLAGS="--sysroot=$NDKROOT/platforms/android-18/arch-mips" \
CFLAGS="--sysroot=$NDKROOT/platforms/android-18/arch-mips" \
./configure --prefix=/data/local/Inst \
--host=mipsel-linux-android --target=mipsel-linux-android \
--with-tmpdir=/sdcard

```

```
# for ARM64 (AArch64)
```

```

CPPFLAGS="--sysroot=$NDKROOT/platforms/android-21/arch-arm64" \
CFLAGS="--sysroot=$NDKROOT/platforms/android-21/arch-arm64" \
./configure --prefix=/data/local/Inst \
--host=aarch64-unknown-linux --target=aarch64-unknown-linux \
--with-tmpdir=/sdcard

```

```

# At the end of the configure run, a few lines of details
# are printed. Make sure that you see these two lines:
#

```

```
# For ARM:
```

```

# Platform variant: android
# Primary -DVGPV string: -DVGPV_arm_linux_android=1
#

```

```
# For x86:
```

```

# Platform variant: android
# Primary -DVGPV string: -DVGPV_x86_linux_android=1
#

```

```
# For mips32:
```

```

# Platform variant: android
# Primary -DVGPV string: -DVGPV_mips32_linux_android=1
#

```

```
# For ARM64 (AArch64):
```

```

# Platform variant: android
# Primary -DVGPV string: -DVGPV_arm64_linux_android=1
#

```

```

# If you see anything else at this point, something is wrong, and
# either the build will fail, or will succeed but you'll get something
# which won't work.

```

```
# Build, and park the install tree in `pwd`/Inst
```

```
#
```

```
make -j4
```

```
make -j4 install DESTDIR=`pwd`/Inst
```

```
# To get the install tree onto the device:
```

```

# (I don't know why it's not "adb push Inst /data/local", but this
# formulation does appear to put the result in /data/local/Inst.)
#

```

```
#
```

```
adb push Inst /
```

---

```

# To run (on the device). There are two things you need to consider:
#
# (1) if you are running on the Android emulator, Valgrind may crash
# at startup. This is because the emulator (for ARM) may not be
# simulating a hardware TLS register. To get around this, run
# Valgrind with:
# --kernel-variant=android-no-hw-tls
#
# (2) if you are running a real device, you need to tell Valgrind
# what GPU it has, so Valgrind knows how to handle custom GPU
# ioctls. You can choose one of the following:
# --kernel-variant=android-gpu-sgx5xx # PowerVR SGX 5XX series
# --kernel-variant=android-gpu-adreno3xx # Qualcomm Adreno 3XX series
# If you don't choose one, the program will still run, but Memcheck
# may report false errors after the program performs GPU-specific ioctls.
#
# Anyway: to run on the device:
#
/data/local/Inst/bin/valgrind [kernel variant args] [the usual args etc]

# Once you're up and running, a handy modify-V-rebuild-reinstall
# command line (on the host, of course) is
#
mq -j2 && mq -j2 install DESTDIR=`pwd`/Inst && adb push Inst /
#
# where 'mq' is an alias for 'make --quiet'.

# One common cause of runs failing at startup is the inability of
# Valgrind to find a suitable temporary directory. On the device,
# there doesn't seem to be any one location which we always have
# permission to write to. The instructions above use /sdcard. If
# that doesn't work for you, and you're Valgrinding one specific
# application which is already installed, you could try using its
# temporary directory, in /data/data, for example
# /data/data/org.mozilla.firefox_beta.
#
# Using /system/bin/logcat on the device is helpful for diagnosing
# these kinds of problems.

```

# 10. README.android\_emulator

How to install and run an android emulator.

```
mkdir android # or any other place you prefer
cd android
```

```
# download java JDK
# http://www.oracle.com/technetwork/java/javase/downloads/index.html
# download android SDK
# http://developer.android.com/sdk/index.html
# download android NDK
# http://developer.android.com/sdk/ndk/index.html
```

```
# versions I used:
# jdk-7u4-linux-i586.tar.gz
# android-ndk-r8-linux-x86.tar.bz2
# android-sdk_r18-linux.tgz
```

```
# install jdk
tar xzf jdk-7u4-linux-i586.tar.gz
```

```
# install sdk
tar xzf android-sdk_r18-linux.tgz
```

```
# install ndk
tar xjf android-ndk-r8-linux-x86.tar.bz2
```

```
# setup PATH to use the installed software:
export SDKROOT=$HOME/android/android-sdk-linux
export PATH=$PATH:$SDKROOT/tools:$SDKROOT/platform-tools
export NDKROOT=$HOME/android/android-ndk-r8
```

```
# install android platforms you want by starting:
android
# (from $SDKROOT/tools)
```

```
# select the platforms you need
# I selected and installed:
#   Android 4.0.3 (API 15)
# Upgraded then to the newer version available:
#   Android sdk 20
#   Android platform tools 12
```

```
# then define a virtual device:
Tools -> Manage AVDs...
# I define an AVD Name with 64 Mb SD Card, (4.0.3, api 15)
# rest is default
```

```
# compile and make install Valgrind, following README.android
```

```
# Start your android emulator (it takes some time).
```

```
# You can use adb shell to get a shell on the device
# and see it is working. Note that I usually get
# one or two time out from adb shell before it works
adb shell
```

```
# Once the emulator is ready, push your Valgrind to the emulator:
adb push Inst /
```

```
# IMPORTANT: when running Valgrind, you may need give it the flag
#
# --kernel-variant=android-no-hw-tls
#
# since otherwise it may crash at startup.
# See README.android for details.
```

```
# if you need to debug:
# You have on the android side a gdbserver
# on the device side:
gdbserver :1234 your_exe
```

```
# on the host side:
adb forward tcp:1234 tcp:1234
$HOME/android/android-ndk-r8/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-gdb your_e
target remote :1234
```

# 11. README.mips

## Supported platforms

-----

- MIPS32 and MIPS64 platforms are currently supported.
- Both little-endian and big-endian cores are supported.
- MIPS DSP ASE on MIPS32 platforms is supported.

## Building V for MIPS

-----

- Native build is available for all supported platforms. The build system expects that native GCC is configured correctly and optimized for the platform. Yet, this may not be the case with some Debian distributions which configure GCC to compile to "mips1" by default. Depending on a target platform, using CFLAGS="-mips32r2", CFLAGS="-mips32" or CFLAGS="-mips64" or CFLAGS="-mips64 -mabi=64" will do the trick and compile Valgrind correctly.

- Use of cross-toolchain is supported as well.
- Example of configure line and additional configure options:

```
$ ./configure --host=mipsel-linux-gnu --prefix=<path_to_install_directory>
```

\* --host=mips-linux-gnu is necessary only if Valgrind is built on platform other than MIPS, tools for building MIPS application have to be in PATH.

\* --host=mips-linux-gnu is necessary if you compile it with cross toolchain compiler for big endian platform.

\* --host=mipsel-linux-musl is necessary if you compile it with cross toolchain compiler for little endian platform.

\* --host=nanomipseb-linux-gnu is necessary if you compile it with cross toolchain compiler for nanoMIPS big endian platform.

\* --host=nanomips-linux-gnu is necessary if you compile it with cross toolchain compiler for nanoMIPS little endian platform.

\* --build=mips-linux is needed if you want to build it for MIPS32 on 64-bit MIPS system.

\* If you are compiling Valgrind for mips32 with gcc version older than gcc (GCC) 4.5.1, you must specify CFLAGS="-mips32r2 -mplt", e.g.

```
./configure --prefix=<path_to_install_directory>  
CFLAGS="-mips32r2 -mplt"
```

## Limitations

-----

- Some gdb tests will fail when gdb (GDB) older than 7.5 is used and gdb is not compiled with '--with-expat=yes'.
- You can not compile tests for DSP ASE if you are using gcc (GCC) older than 4.6.1 due to a bug in the toolchain.
- Older GCC may have issues with some inline assembly blocks. Get a toolchain

based on newer GCC versions, if possible.

- Systems with a mips64 cpu having only o32 libraries will misconfigure in case no appropriate architecture flag is specified during configure time.  
Be sure to set either mips32 or mips32r2 as the target architecture in that case.
- Some tests can not be compiled for nanoMIPS due to limitations in preliminary GCC for nanoMIPS. You can use '-i' switch for building tests.

# 12. README.solaris

## Requirements

- You need a recent Solaris-like OS to compile this port. Solaris 11 or any illumos-based distribution should work, Solaris 10 is not supported. Running ``uname -r`` has to print '5.11'.
- Recent GCC tools are required, GCC 3 will probably not work. GCC version 4.5 (or higher) is recommended.
- Solaris ld has to be the first linker in the PATH. GNU ld cannot be used. There is currently no linker check in the configure script but the linking phase fails if GNU ld is used. Recent Solaris/illumos distributions are ok.
- A working combination of autotools is required: aclocal, autoheader, automake and autoconf have to be found in the PATH. You should be able to install `pkg:/developer/build/automake` and `pkg:/developer/build/autoconf` packages to fulfil this requirement.
- System header files are required. On Solaris, these can be installed with:  
`# pkg install system/header`
- GNU make is also required. On Solaris, this can be quickly achieved with:  
`$ PATH=/usr/gnu/bin:$PATH; export PATH`
- For remote debugging support, working GDB is required (see below).
- For running regression tests, GNU sed, grep, awk, diff are required. This can be quickly achieved on Solaris by prepending `/usr/gnu/bin` to PATH.

## Compilation

Please follow the generic instructions in the README file, in the section 'Building and installing it'.

The configure script detects a canonical host to determine which version of Valgrind should be built. If the system compiler by default produces 32-bit binaries then only a 32-bit version of Valgrind will be built. To enable compilation of both 64-bit and 32-bit versions on such a system, issue the configure script as follows:

```
./configure CC='gcc -m64' CXX='g++ -m64'
```

## Oracle Solaris and illumos support

One of the main goal of this port is to support both Oracle Solaris and illumos kernels. This is a very hard task because Solaris kernel traditionally does not provide a stable syscall interface and because Valgrind contains several parts that are closely tied to the underlying kernel. For these reasons, the port needs to detect which syscall interfaces are present. This detection cannot be done easily at run time and is currently implemented as a set of configure tests. This means that a binary version of this port can be executed only on a kernel that is compatible with a kernel that was used during the configure and compilation time.

Main currently-known incompatibilities:

- Solaris 11 (released in November 2011) removed a large set of syscalls where \*at variant of the syscall was also present, for example, `open()` versus `openat(AT_FDCWD)` [1]
- syscall number for `unlinkat()` is 76 on Solaris 11, but 65 on illumos [2]
- illumos (in April 2013) changed interface of the `accept()` and `pipe()`

syscalls [3]

- posix\_spawn() functionality is backed up by true spawn() syscall on Solaris 11.4 whereas illumos and Solaris 11.3 leverage vfork()
- illumos and older Solaris use utimesys() syscall whereas newer Solaris uses utimensat()

[1] [http://docs.oracle.com/cd/E26502\\_01/html/E28556/gkzlf.html#gkzip](http://docs.oracle.com/cd/E26502_01/html/E28556/gkzlf.html#gkzip)

[2] <https://www.illumos.org/issues/521>

[3] <https://github.com/illumos/illumos-gate/commit/5dbfd19ad5fcc2b779f40f80fa05c1bd28fd0b4e>

## Limitations

-----

- The port is Work-In-Progress, many things may not work or they can be subtly broken.
- Coredumps produced by Valgrind do not contain all information available, especially microstate accounting and processor bindings.
- Accessing contents of /proc/self/psinfo is not thread-safe. That is because Valgrind emulates this file on behalf of the client programs. Entire open() - read() - close() sequence on this file needs to be performed atomically.
- Fork limitations: vfork() is translated to fork(), forkall() is not supported.
- Valgrind does not track definedness of some eflags (OF, SF, ZF, AF, CF, PF) individually for each flag. After a syscall is finished, when a carry flag is set and defined, all other mentioned flags will be also defined even though they might be undefined before making the syscall.
- System call "execve" with a file descriptor which points to a hardlink is currently not supported. That is because from the opened file descriptor itself it is not possible to reverse map the intended pathname. Examples are fexecve(3C) and isaexec(3C).
- Program headers PT\_SUNW\_SYSSTAT and PT\_SUNW\_SYSSTAT\_ZONE are not supported. That is, programs linked with mapfile directive RESERVE\_SEGMENT and attribute TYPE equal to SYSSTAT or SYSSTAT\_ZONE will cause Valgrind exit. It is not possible for Valgrind to arrange mapping of a kernel shared page at the address specified in the mapfile for the guest application. There is currently no such mechanism in Solaris. Hacky workarounds are possible, though.
- When a thread has no stack then all system calls will result in Valgrind crash, even though such system calls use just parameters passed in registers. This should happen only in pathological situations when a thread is created with custom mmap'ed stack and this stack is then unmap'ed during thread execution.

## Remote debugging support

-----

Solaris port of GDB has a major flaw which prevents remote debugging from working correctly. Fortunately this flaw has an easy fix [4]. Unfortunately it is not present in the current GDB 7.6.2. This boils down to several options:

- Use GDB shipped with Solaris 11.2 which has this flaw fixed.
- Wait until GDB 7.7 becomes available (there won't be other 7.6.x releases).
- Build GDB 7.6.2 with the fix by yourself using the following steps:

```
# pkg install developer/gnu-binutils
$ wget http://ftp.gnu.org/gnu/gdb/gdb-7.6.2.tar.gz
$ gzip -dc gdb-7.6.2.tar.gz | tar xf -
$ cd gdb-7.6.2
$ patch -p1 -i /path/to/valgrind-solaris/solaris/gdb-sol-thread.patch
```

```
$ export LIBS="-lncurses"
$ export CC="gcc -m64"
$ ./configure --with-x=no --with-curses --with-libexpat-prefix=/usr/lib
$ gmake && gmake install
```

[4] <https://sourceware.org/ml/gdb-patches/2013-12/msg00573.html>

#### TODO list

-----

- Fix few remaining failing tests.
- Add more Solaris-specific tests (especially for the door and spawn syscalls).
- Provide better error reporting for various subsyscalls.
- Implement storing of extra register state in signal frame.
- Performance comparison against other platforms.
- Prevent SIGPIPE when writing to a socket (coregrind/m\_libcfile.c).
- Implement ticket locking for fair scheduling (--fair-sched=yes).
- Implement support in DRD and Helgrind tools for thr\_join() with thread == 0.
- Add support for accessing thread-local variables via gdb (auxprogs/getoff.c). Requires research on internal libc TLS representation.
- VEX supports AVX, BMI and AVX2. Investigate if they can be enabled on Solaris/illumos.
- Investigate support for more flags in AT\_SUN\_AUXFLAGS.
- Fix Valgrind crash when a thread has no stack and syswrap-main.c accesses all possible syscall parameters. Enable helgrind/tests/stackteardown.c to see this in effect. Would require awareness of syscall parameter semantics.
- Correctly print arguments of DW\_CFA\_ORCL\_arg\_loc in show\_CF\_instruction() when it is implemented in libdwarf.
- Handle a situation when guest program sets SC\_CANCEL\_FLG in schedctl and Valgrind needs to invoke a syscall on its own.

#### Summary of Solaris 11 Kernel Interfaces Used

-----

Valgrind uses directly the following kernel interfaces (not exhaustive list). Then, of course, it has very intimate knowledge of all syscalls, many ioctls and some door calls because it has wrappers around them.

- Syscalls:
  - . clock\_gettime
  - . close
  - . connect
  - . execve
  - . exit
  - . faccessat
  - . fcntl
  - . forksys
  - . fstatat
  - . getcwd
  - . getdents
  - . geteuid
  - . getgid
  - . getgroups
  - . getpeername
  - . getpid
  - . getrlimit
  - . getsockname
  - . getsockopt

- . gettimeofday
- . kill
- . lseek
- . lwp\_create
- . lwp\_exit
- . lwp\_self
- . lwp\_sigqueue
- . mknodat
- . mmap
- . mprotect
- . munmap
- . openat
- . pipe
- . pollsys
- . pread
- . prgpsys
- . pwrite
- . read
- . readlinkat
- . renameat
- . rt\_sigprocmask
- . send
- . setrlimit
- . setsockopt
- . sigaction
- . sigreturn
- . sigtimedwait
- . so\_socket
- . spawn
- . uname
- . unlinkat
- . waitsys
- . write

- Signal frames. Valgrind decomposes and synthesizes signal frames.
- Flag `sc_sigblock` flag in the `schedctl` structure by replacing function `block_all_signals()` from `libc`. The replacement emulates `lwp_sigmask` syscall. More details in `coregrind/vg_preloaded.c`.
- Initial stack layout for the main thread is synthesized.
- `procfs` agent thread and other `procfs` commands for manipulating the process.
- `mmapobj` syscall is emulated because it gets in the way of the address space manager's control.

## Contacts

-----

Please send bug reports and any questions about the port to:

Ivo Rair [<ivosh@ivosh.net>](mailto:ivosh@ivosh.net)

Petr Pavlu [<setup@dagobah.cz>](mailto:setup@dagobah.cz)

# 13. README.freebsd

Installing from ports or via pkg

~~~~~

You can install Valgrind using either

pkg install devel/valgrind

or alternatively from ports (if installed)

cd /usr/ports/devel/valgrind && make install clean

devel/valgrind is updated with official releases of Valgrind, normally in April and October each year. There is an alternative port, devel/valgrind-devel which occasionally gets updated from the latest Valgrind source. If you want to have the latest port, check on <https://www.freshports.org/> to see which is the most recent. If you want to have the very latest version, you will need to build a copy from source. See README for instructions on getting the source with git.

Building Valgrind

~~~~~

Install ports for autotools, gmake and python.

```
$ sh autogen.sh
$ ./configure --prefix=/where/ever
$ gmake
$ gmake install
```

If you are using a jail for building, make sure that it is configured so that "uname -r" returns a string that matches the pattern "XX.Y-\*" where XX is the major version (12, 13, 14 ...) and Y is the minor version (0, 1, 2, 3).

Known Limitations (June 2022)

0. Be aware that if you use a wrapper script and run Valgrind on the wrapper script Valgrind may hit restrictions if the wrapper script runs any Capsicum enabled applications. Examples of Capsicum enabled applications are echo, basename, tee, uniq and wc. It is recommended that you either avoid these applications or that you run Valgrind directly on your test application.
1. There are some limitations when running Valgrind on code that was compiled with clang. These issues are not present with code compiled with GCC.
  - a) There may be missing source information concerning variables due to DWARF extensions used by GCC.
  - b) Code that uses OpenMP will generate spurious errors.
2. vgdb invoker, which uses ptrace, may cause system calls to be interrupted. As an example, if the debuggee seems to have be stuck and you press Ctrl-C in gdb the debuggee may execute one more statement before stopping and returning control to gdb.

Notes for Developers

~~~~~  
See README\_DEVELOPERS, README\_MISSING\_SYSCALL\_OR\_IOCTL and docs/\*  
for more general information for developers.

## 0. Adding syscalls.

When adding syscalls, you need to look at the manpage and also syscalls.master  
(online at  
<https://github.com/freebsd/freebsd/blob/master/sys/kern/syscalls.master>  
and for 32bit  
<https://github.com/freebsd/freebsd/blob/master/sys/compat/freebsd32/syscalls.master>

and if you installed the src package there should also be

```
/usr/src/sys/kern/syscalls.master  
and  
/usr/src/sys/compat/freebsd32/syscalls.master)
```

syscalls.master is particularly useful for seeing quickly whether parameters  
are inputs or outputs.

The syscall wrappers can vary from trivial to difficult. Fortunately, many are  
either trivial (no arguments) or easy (Valgrind just needs to know what memory  
is being read or written). Some syscalls, such as those involving process  
creation and termination, signals and memory mapping require deeper interaction  
with Valgrind.

When you add syscalls you will need to modify several files

- a) include/vki/vki-scnums-freebsd.h  
This file contains one #define for each syscall. The \_NR\_ prefix (Linux  
style) is used rather than SYS\_ for compatibility with the rest of the  
Valgrind source.
- b) coregrind/m\_syswrap/priv\_syswrap-freebsd.h  
This uses the DECL\_TEMPLATE macro to generate declarations for the syscall  
before and after wrappers.
- c) coregrind/m\_syswrap/syswrap-freebsd.c  
This is where the bulk of the code resides. Toward the end of the file  
the BSDX\_/BSDXY macros are used to generate entries in the table of  
syscalls. BSDX\_ is used for wrappers that only have a 'before', BSDXY  
if both wrappers are required. In general, syscalls that have no arguments  
or only input arguments just need a BSDX\_ macro (before only). Syscalls  
with output arguments need a BSDXY macro (before and after).
- d) If the syscall uses 64bit arguments (long long) then instead of putting  
the wrapper definitions in syswrap-freebsd.c there will be one definition  
for each platform amd64 and x86 in syswrap-x86-freebsd.c and  
syswrap-amd64-freebsd.c.  
Each long long needs to be split into two ARGs in the x86 version.

The PRE (before) wrapper

-----

Each PRE wrapper always contains the following two macro calls

PRINT. This outputs the syscall name and argument values when Valgrind is  
executed with  
--trace-syscalls=yes

PRE\_READ\_REGX. This macro lets Valgrind know about the number and types of the syscall arguments which allows Valgrind to check that they are initialized. X is the number of arguments. It is best that the argument names match the man page, but they must match the types and number of arguments in syscalls.master. Occasionally there are differences between the two.

If the syscall takes pointers to memory there will be one of the following for each pointer argument.

PRE\_MEM\_RASCIIZ for NULL terminated ascii strings.

PRE\_MEM\_READ for pointers to structures or arrays that are read.

PRE\_MEM\_WRITE for pointers to structures or arrays that are written.

As a rule, the definitions of structures are copied into vki-freebsd.h with the vki- prefix. [vki - Valgrind kernel interface; this was done historically to protect against discrepancies between user include structure definitions and kernel definitions on Linux].

The POST (after) wrapper

-----

These are much easier.

They just contain a POST\_MEM\_WRITE macro for each output argument.

## 1. Frequent causes of problems

- New \_umtx\_op codes. Valgrind will print "WARNING: \_umtx\_op unsupported value". See syswrap-freebsd.c and add new cases for the new codes.
- Additions to auxv. Depending on the entry it may need to be simply copied from the host to the guest, it may need to be modified for the guest or it may need to be ignored. See initimg-freebsd.c.
- ELF PT\_LOAD mappings. Either Valgrind will assert or there will be no source information in error reports. See VG\_(di\_notify\_mmap) in debuginfo.c
- Because they contain many deliberate errors the regression tests are prone to change with changes of compiler. Liberal use of 'volatile' and '-Wno-warning-flag' can help - see configure.ac

## 2. Running regression tests

In order to run all of the regression tests you will need to install the following packages

gdb  
gsed

In addition to running "gmake" you will need to run "gmake check" to build the regression test executables and "gmake regtest". Again, more details can be seen in README\_DEVELOPERS.

If you want to run the 'nightly' script (see nightly/README.txt) you will need to install coreutils (for GNU cp) and modify the nightly/conf/freebsd.\* files. The default configuration sends an e-mail to the valgrind-testresults mailing list.

Feedback

~~~~~

If you find any problems please create a bugzilla report at <https://bugs.kde.org> using the Valgrind product.

Alternatively you can use the FreeBSD bugilla <https://bugs.freebsd.org>

#### Credits

~~~~~

Valgrind was originally ported to FreeBSD by Doug Rabson in 2004.

Paul Floyd (that's me), started looking at this project in late 2018, took a long pause and then continued in earnest in January 2020.

A big thanks to Nick Briggs for helping with the x86 version.

Kyle Evans and Ed Maste for contributing patches and helping with the integration with FreeBSD ports.

Prior to 2018 many others have also contributed.

Dimitry Andric  
Simon Barner  
Roman Bogorodskiy  
Rebecca Cran  
Bryan Drewery  
Brian Fundakowski Feldman  
Denis Generalov  
Mikolaj Golub  
Eugene Kilachkoff  
Xin LI  
Phil Longstaff  
Pav Lucistnik  
Conrad Meyer  
Julien Nadeau  
Frerich Raabe  
Doug Rabson  
Craig Rodrigues  
Tom Russo  
Stephen Sanders  
Stanislav Sedov  
Andrei V. Shetuhin  
Niklas Sorensson  
Ryan Stone  
Jerry Toungh  
Yuri

# GNU Licenses

## Table of Contents

1. The GNU General Public License .....	1
2. The GNU Free Documentation License .....	7

# 1. The GNU General Public License

GNU GENERAL PUBLIC LICENSE  
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,  
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

## Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE  
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

#### NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

#### END OF TERMS AND CONDITIONS

#### How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>  
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author  
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.  
This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program  
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989  
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

# 2. The GNU Free Documentation License

GNU Free Documentation License  
Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.  
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of

the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements",

"Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an

Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

#### 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers

- or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

### ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

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

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.