Valgrind Documentation

Release 3.0.1 August 29 2005

Copyright © 2000-2005 AUTHORS

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled The GNU Free Documentation License [11].

Table of Contents

Valgrind Quick Start Guide . . . 1
Valgrind User Manual . . . 1
- 1. Introduction . . . 1
  - 1.1. An Overview of Valgrind . . . 1
  - 1.2. How to navigate this manual . . . 2
- 2. Using and understanding the Valgrind core . . . 4
  - 2.1. What Valgrind does with your program . . . 4
  - 2.2. Getting started . . . 5
  - 2.3. The commentary . . . 6
  - 2.4. Reporting of errors . . . 7
  - 2.5. Suppressing errors . . . 8
  - 2.6. Command-line flags for the Valgrind core . . . 10
    - 2.6.1. Tool-selection option . . . 10
    - 2.6.2. Basic Options . . . 11
    - 2.6.3. Error-related options . . . 12
    - 2.6.4. malloc()-related Options . . . 15
    - 2.6.5. Uncommon Options . . . 15
    - 2.6.6. Debugging Valgrind Options . . . 17
    - 2.6.7. Setting default Options . . . 17
  - 2.7. The Client Request mechanism . . . 17
  - 2.8. Support for Threads . . . 20
  - 2.9. Handling of Signals . . . 20
  - 2.10. Building and Installing . . . 20
  - 2.11. If You Have Problems . . . 21
  - 2.12. Limitations . . . 21
  - 2.13. How It Works -- A Rough Overview . . . 23
    - 2.13.1. Getting started . . . 23
    - 2.13.2. The translation/instrumentation engine . . . 24
    - 2.13.3. Tracking the Status of Memory . . . 24
    - 2.13.4. System calls . . . 24
    - 2.13.5. Signals . . . 25
  - 2.14. An Example Run . . . 25
  - 2.15. Warning Messages You Might See . . . 26
- 3. Memcheck: a heavyweight memory checker . . . 28
  - 3.1. Kinds of bugs that Memcheck can find . . . 28
  - 3.2. Command-line flags specific to Memcheck . . . 28
  - 3.3. Explanation of error messages from Memcheck . . . 30
    - 3.3.1. Illegal read / Illegal write errors . . . 30
    - 3.3.2. Use of uninitialised values . . . 31
    - 3.3.3. Illegal frees . . . 32
    - 3.3.4. When a block is freed with an inappropriate deallocation function . . . 32
    - 3.3.5. Passing system call parameters with inadequate read/write permissions . . . 33
    - 3.3.6. Overlapping source and destination blocks . . . 34
    - 3.3.7. Memory leak detection . . . 34
  - 3.4. Writing suppression files . . . 35
  - 3.5. Details of Memcheck's checking machinery . . . 36
    - 3.5.1. Valid-value (V) bits . . . 36
    - 3.5.2. Valid-address (A) bits . . . 37
    - 3.5.3. Putting it all together . . . 38
  - 3.6. Client Requests . . . 39
- 4. Addrcheck: a lightweight memory checker . . . 41
  - 4.1. Kinds of bugs that Addrcheck can find . . . 41

5. Cachegrind: a cache profiler 43
  5.1. Cache profiling 43
    5.1.1. Overview 43
    5.1.2. Cache simulation specifics 44
  5.2. Profiling programs 45
    5.2.1. Output file 45
    5.2.2. Cachegrind options 46
    5.2.3. Annotating C/C++ programs 46
    5.2.4. Annotating assembler programs 51
  5.3. cg_annotate options 51
    5.3.1. Warnings 52
    5.3.2. Things to watch out for 53
    5.3.3. Accuracy 54
    5.3.4. Todo 54
6. Massif: a heap profiler 55
  6.1. Heap profiling 55
    6.1.1. Why Use a Heap Profiler? 56
  6.2. Using Massif 56
    6.2.1. Overview 56
    6.2.2. Basic Results of Profiling 56
    6.2.3. Spacetime Graphs 57
  6.3. Details of Heap Allocations 58
    6.3.1. Accuracy 60
  6.4. Massif options 60
7. Helgrind: a data-race detector 62
8. Nulgrind: the “null” tool 63
9. Lackey: a very simple profiler 64
Valgrind FAQ 1
Valgrind Technical Documentation 1
1. The Design and Implementation of Valgrind 5
  1.1. Introduction 5
    1.1.1. History 6
    1.1.2. Design overview 6
    1.1.3. Design decisions 7
    1.1.4. Correctness 8
    1.1.5. Current limitations 10
  1.2. The instrumenting JITter 10
    1.2.1. Run-time storage, and the use of host registers 10
    1.2.2. Startup, shutdown, and system calls 12
    1.2.3. Introduction to UCode 12
    1.2.4. UCode operand tags: type Tag 13
    1.2.5. UCode instructions: type UInstr 14
    1.2.6. Translation into UCode 17
    1.2.7. UCode optimisation 19
    1.2.8. UCode instrumentation 20
    1.2.9. UCode post-instrumentation cleanup 23
    1.2.10. Translation from UCode 25
    1.2.11. Top-level dispatch loop 26
    1.2.12. Lazy updates of the simulated program counter 26
    1.2.13. Signals 26
    1.2.14. To be written 26
  1.3. Extensions 27
    1.3.1. Bugs 27
    1.3.2. Threads 27

1.3.3. Verification suite 27
1.3.4. Porting to other platforms 27
1.4. Easy stuff which ought to be done 28
1.4.1. MMX Instructions 28
1.4.2. Fix stabs-info reader 28
1.4.3. BT/BTC/BTS/BTR 28
1.4.4. Using PREFETCH Instructions 28
1.4.5. User-defined Permission Ranges 29
2. How Cachegrind works 32
2.1. Cache profiling 32
2.2. Cost centres 32
2.3. Storing cost-centres 34
2.4. Instrumentation 35
2.5. Handling basic block retranslations 37
2.6. The cache simulation 38
2.7. Output 38
2.8. Summary of performance features 39
2.9. Annotation 39
2.10. Similar work, extensions 39
3. Writing a New Valgrind Tool 40
3.1. Introduction 40
3.1.1. Supervised Execution 40
3.1.2. Tools 40
3.1.3. Execution Spaces 41
3.2. Writing a Tool 41
3.2.1. Why write a tool? 41
3.2.2. Suggested tools 42
3.2.3. How tools work 43
3.2.4. Getting the code 43
3.2.5. Getting started 43
3.2.6. Writing the code 45
3.2.7. Initialisation 45
3.2.8. Instrumentation 45
3.2.9. Finalisation 46
3.2.10. Other Important Information 46
3.2.11. Words of Advice 46
3.3. Advanced Topics 47
3.3.1. Suppressions 47
3.3.2. Documentation 47
3.3.3. Regression Tests 49
3.3.4. Profiling 50
3.3.5. Other Makefile Hackery 50
3.3.6. Core/tool Interface Versions 50
3.4. Final Words 50
Distribution Documents 1
1. ACKNOWLEDGEMENTS 4
2. AUTHORS 6
3. INSTALL 7
4. NEWS 11
5. README 32
6. README_MISSING_SYSCALL_OR_IOCTL 34
7. README_DEVELOPERS 38
8. README_PACKAGERS 40
GNU Licenses 1

1. The GNU General Public License 4
2. The GNU Free Documentation License 11

Valgrind Quick Start Guide

Valgrind Quick Start Guide

The Valgrind distribution has multiple tools. The most popular is the memory checking tool (called Memcheck) which can detect many common memory errors such as:

- touching memory you shouldn't (eg. overrunning heap block boundaries);
- using values before they have been initialized;
- incorrect freeing of memory, such as double-freeing heap blocks;
- memory leaks;

What follows is the minimum information you need to start detecting memory errors in your program with Memcheck. Note that this guide applies to Valgrind version 2.4.0 and later; some of the information is not quite right for earlier versions.

1. Preparing your program

Compile your program with `-g` to include debugging information so that Memcheck's error messages include exact line numbers.

2. Running your program under Memcheck

If you normally run your program like this:

```
myprog arg1 arg2
```

Use this command line:

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

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

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

3. Interpreting Memcheck's output

Here's an example C program with a memory error and a memory leak.

```
#include <stdlib.h>

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

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

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

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

Things to notice:

- There is a lot of information in each error message; read it carefully.
- The 19182 is the process ID; it's usually unimportant.
- The first line ("Invalid write...") tells you what kind of error it is. Here, the program wrote to some memory it should not have due to a heap block overrun.
- Below the first line is a stack trace telling you where the problem occurred. Stack traces can get quite large, and be confusing, especially if you are using the C++ STL. Reading them from the bottom up can help. If the stack trace is not big enough, use the `--num-callers` option to make it bigger.
- The addresses (eg. 0x804838F) are usually unimportant, but occasionally crucial for tracking down weirder bugs.
- Some error messages have a second component which describes the memory address involved. This one shows that the written memory is just past the end of a block allocated with malloc() on line 7 of example.c.

It's worth fixing errors in the order they are reported, as later errors can be caused by earlier errors.

Memory leak messages look like this:

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:7)
==19182==    by 0x80483AB: main (a.c:14)
```

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

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

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

If you don't understand an error message, please consult Explanation of error messages from Memcheck [30] in the Valgrind User Manual [1] which has examples of all the error messages Memcheck produces.

4. Caveats

Memcheck is not perfect; it occasionally produces false positives, and there are mechanisms for suppressing these (see Suppressing errors [8] in the Valgrind User Manual [1]). However, it is typically right 99% of the time, so you should be wary of ignoring its error messages. After all, you wouldn't ignore warning messages produced by a compiler, right?

Memcheck also cannot detect every memory error your program has. For example, it can't detect if you overrun the bounds of an array that is allocated statically or on the stack.

5. More information

Please consult the Valgrind FAQ [1] and the Valgrind User Manual [1], which have much more information. Note that the other tools in the Valgrind distribution can be invoked with the `--tool` option.

Valgrind User Manual

Valgrind User Manual

Table of Contents

1. Introduction . . . 1
    - 1.1. An Overview of Valgrind . . . 1
    - 1.2. How to navigate this manual . . . 2

2. Using and understanding the Valgrind core . . . 4
    - 2.1. What Valgrind does with your program . . . 4
    - 2.2. Getting started . . . 5
    - 2.3. The commentary . . . 6
    - 2.4. Reporting of errors . . . 7
    - 2.5. Suppressing errors . . . 8
    - 2.6. Command-line flags for the Valgrind core . . . 10
        - 2.6.1. Tool-selection option . . . 10
        - 2.6.2. Basic Options . . . 11
        - 2.6.3. Error-related options . . . 12
        - 2.6.4. malloc()-related Options . . . 15
        - 2.6.5. Uncommon Options . . . 15
        - 2.6.6. Debugging Valgrind Options . . . 17
        - 2.6.7. Setting default Options . . . 17
    - 2.7. The Client Request mechanism . . . 17
    - 2.8. Support for Threads . . . 20
    - 2.9. Handling of Signals . . . 20
    - 2.10. Building and Installing . . . 20
    - 2.11. If You Have Problems . . . 21
    - 2.12. Limitations . . . 21
    - 2.13. How It Works -- A Rough Overview . . . 23
        - 2.13.1. Getting started . . . 23
        - 2.13.2. The translation/instrumentation engine . . . 24
        - 2.13.3. Tracking the Status of Memory . . . 24
        - 2.13.4. System calls . . . 24
        - 2.13.5. Signals . . . 25
    - 2.14. An Example Run . . . 25
    - 2.15. Warning Messages You Might See . . . 26

3. Memcheck: a heavyweight memory checker . . . 28
    - 3.1. Kinds of bugs that Memcheck can find . . . 28
    - 3.2. Command-line flags specific to Memcheck . . . 28
    - 3.3. Explanation of error messages from Memcheck . . . 30
        - 3.3.1. Illegal read / Illegal write errors . . . 30
        - 3.3.2. Use of uninitialised values . . . 31
        - 3.3.3. Illegal frees . . . 32
        - 3.3.4. When a block is freed with an inappropriate deallocation function . . . 32
        - 3.3.5. Passing system call parameters with inadequate read/write permissions . . . 33
        - 3.3.6. Overlapping source and destination blocks . . . 34
        - 3.3.7. Memory leak detection . . . 34
    - 3.4. Writing suppression files . . . 35
    - 3.5. Details of Memcheck's checking machinery . . . 36
        - 3.5.1. Valid-value (V) bits . . . 36
        - 3.5.2. Valid-address (A) bits . . . 37
        - 3.5.3. Putting it all together . . . 38
    - 3.6. Client Requests . . . 39

4. Addrcheck: a lightweight memory checker . . . 41
    - 4.1. Kinds of bugs that Addrcheck can find . . . 41

5. Cachegrind: a cache profiler . . . 43
    - 5.1. Cache profiling . . . 43

        - 5.1.1. Overview 43
        - 5.1.2. Cache simulation specifics 44
    - 5.2. Profiling programs 45
        - 5.2.1. Output file 45
        - 5.2.2. Cachegrind options 46
        - 5.2.3. Annotating C/C++ programs 46
        - 5.2.4. Annotating assembler programs 51
    - 5.3. cg_annotate options 51
        - 5.3.1. Warnings 52
        - 5.3.2. Things to watch out for 53
        - 5.3.3. Accuracy 54
        - 5.3.4. Todo 54
- 6. Massif: a heap profiler 55
    - 6.1. Heap profiling 55
        - 6.1.1. Why Use a Heap Profiler? 56
    - 6.2. Using Massif 56
        - 6.2.1. Overview 56
        - 6.2.2. Basic Results of Profiling 56
        - 6.2.3. Spacetime Graphs 57
    - 6.3. Details of Heap Allocations 58
        - 6.3.1. Accuracy 60
    - 6.4. Massif options 60
- 7. Helgrind: a data-race detector 62
- 8. Nulgrind: the “null” tool 63
- 9. Lackey: a very simple profiler 64

1. Introduction

Table of Contents

1.1. An Overview of Valgrind 1
1.2. How to navigate this manual 2

1.1. An Overview of Valgrind

Valgrind is a flexible system for debugging and profiling Linux executables. The system consists of a core, which provides a synthetic CPU in software, and a series of tools, each of which performs some kind of debugging, profiling, or similar task. The architecture is modular, so that new tools can be created easily and without disturbing the existing structure.

A number of useful tools are supplied as standard. In summary, these are:

1. **Memcheck** detects memory-management problems in your programs. All reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted. As a result, Memcheck can detect the following problems:

    - Use of uninitialised memory
    - Reading/writing memory after it has been free'd
    - Reading/writing off the end of malloc'd blocks
    - Reading/writing inappropriate areas on the stack
    - Memory leaks -- where pointers to malloc'd blocks are lost forever
    - Mismatched use of malloc/new/new [] vs free/delete/delete []
    - Overlapping `src` and `dst` pointers in `memcpy()` and related functions
    - Some misuses of the POSIX pthreads API

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

2. **Addrcheck** is a lightweight version of Memcheck. It is identical to Memcheck except for the single detail that it does not do any uninitialised-value checks. All of the other checks -- primarily the fine-grained address checking -- are still done. The downside of this is that you don't catch the uninitialised-value errors that Memcheck can find.

   But the upside is significant: programs run about twice as fast as they do on Memcheck, and a lot less memory is used. It still finds reads/writes of freed memory, memory off the end of blocks and in other invalid places, bugs which you really want to find before release!

   Because Addrcheck is lighter and faster than Memcheck, you can run more programs for longer, and so you may be able to cover more test scenarios. Addrcheck was created because one of us (Julian) wanted to be able to run a complete KDE desktop session with checking. As of early November 2002, we have been able to run KDE-3.0.3 on a 1.7 GHz P4 with 512 MB of memory, using Addrcheck. Although the result is not stellar, it's quite usable, and it seems plausible to run KDE for long periods at a time like this, collecting up all the addressing errors that appear.

3. **Cachegrind** is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code. If you desire, it will show the number of cache misses, memory references and instructions accruing to each line of source code, with per-function, per-module and whole-program summaries. If you ask really nicely it will even show counts for each individual machine instruction.

   On x86 and AMD64, Cachegrind auto-detects your machine's cache configuration using the `CPUID` instruction, and so needs no further configuration info, in most cases.

   Cachegrind is nicely complemented by Josef Weidendorfer's amazing KCacheGrind visualisation tool (http://kcachegrind.sourceforge.net), a KDE application which presents these profiling results in a graphical and easier-to-understand form.

4. **Helgrind** finds data races in multithreaded programs. Helgrind looks for memory locations which are accessed by more than one (POSIX p-)thread, but for which no consistently used (pthread_mutex_)lock can be found. Such locations are indicative of missing synchronisation between threads, and could cause hard-to-find timing-dependent problems.

   Helgrind ("Hell's Gate", in Norse mythology) implements the so-called "Eraser" data-race-detection algorithm, along with various refinements (thread-segment lifetimes) which reduce the number of false errors it reports. It is as yet somewhat of an experimental tool, so your feedback is especially welcomed here.

   Helgrind has been hacked on extensively by Jeremy Fitzhardinge, and we have him to thank for getting it to a releasable state.

A number of minor tools (**Corecheck**, **Lackey** and **Nulgrind**) are also supplied. These aren't particularly useful -- they exist to illustrate how to create simple tools and to help the valgrind developers in various ways.

Valgrind is closely tied to details of the CPU and operating system, and to a lesser extent, the compiler and basic C libraries. Nonetheless, as of version 3.0.0 it supports several platforms: x86/Linux (mature), AMD64/Linux (immature but works well), and PPC32/Linux (very preliminary). Valgrind uses the standard Unix `./configure`, `make`, `make install` mechanism, and we have attempted to ensure that it works on machines with kernel 2.4 or 2.6 and glibc 2.2.X--2.4.X.

Valgrind is licensed under the The GNU General Public License [4], version 2. The `valgrind/*.h` headers that you may wish to include in your code (eg. `valgrind.h`, `memcheck.h`) are distributed under a BSD-style license, so you may include them in your code without worrying about license conflicts. Some of the PThreads test cases, `pth_*.c`, are taken from "Pthreads Programming" by Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrell, ISBN 1-56592-115-1, published by O'Reilly & Associates, Inc.

1.2. How to navigate this manual

The Valgrind distribution consists of the Valgrind core, upon which are built Valgrind tools, which do different kinds of debugging and profiling. This manual is structured similarly.

First, we describe the Valgrind core, how to use it, and the flags it supports. Then, each tool has its own chapter in this manual. You only need to read the documentation for the core and for the tool(s) you actually use, although you may find it helpful to be at least a little bit familar with what all tools do. If you're new to all this, you probably want to run the Memcheck tool. If you want to write a new tool, read Writing a New Valgrind Tool [40].

Be aware that the core understands some command line flags, and the tools have their own flags which they know about. This means there is no central place describing all the flags that are accepted -- you have to read the flags documentation both for Valgrind's core [4] and for the tool you want to use.

2. Using and understanding the Valgrind core

Table of Contents

2.1. What Valgrind does with your program 4
2.2. Getting started 5
2.3. The commentary 6
2.4. Reporting of errors 7
2.5. Suppressing errors 8
2.6. Command-line flags for the Valgrind core 10
    2.6.1. Tool-selection option 10
    2.6.2. Basic Options 11
    2.6.3. Error-related options 12
    2.6.4. malloc()-related Options 15
    2.6.5. Uncommon Options 15
    2.6.6. Debugging Valgrind Options 17
    2.6.7. Setting default Options 17
2.7. The Client Request mechanism 17
2.8. Support for Threads 20
2.9. Handling of Signals 20
2.10. Building and Installing 20
2.11. If You Have Problems 21
2.12. Limitations 21
2.13. How It Works -- A Rough Overview 23
    2.13.1. Getting started 23
    2.13.2. The translation/instrumentation engine 24
    2.13.3. Tracking the Status of Memory 24
    2.13.4. System calls 24
    2.13.5. Signals 25
2.14. An Example Run 25
2.15. Warning Messages You Might See 26

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

2.1. What Valgrind does with your program

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

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

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

(Memcheck is the default, so if you want to use it you can actually omit the `--tool` flag.

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

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

The amount of instrumentation code added varies widely between tools. At one end of the scale, Memcheck adds code to check every memory access and every value computed, increasing the size of the code at least 12 times, and making it run 25-50 times slower than natively. At the other end of the spectrum, the ultra-trivial "none" tool (a.k.a. Nulgrind) adds no instrumentation at all and causes in total "only" about a 4 times slowdown.

Valgrind simulates every single instruction your program executes. Because of this, the active tool checks, or profiles, not only the code in your application but also in all supporting dynamically-linked (`.so`-format) libraries, including the GNU C library, the X client libraries, Qt, if you work with KDE, and so on.

If you're using one of the error-detection tools, Valgrind will often detect errors in libraries, for example the GNU C or X11 libraries, which you have to use. You might not be interested in these errors, since you probably have no control over that code. Therefore, Valgrind allows you to selectively suppress errors, by recording them in a suppressions file which is read when Valgrind starts up. The build mechanism attempts to select suppressions which give reasonable behaviour for the libc and XFree86 versions detected on your machine. To make it easier to write suppressions, you can use the `--gen-suppressions=yes` option which tells Valgrind to print out a suppression for each error that appears, which you can then copy into a suppressions file.

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

2.2. Getting started

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

Another flag you might like to consider, if you are working with C++, is `-fno-inline`. That makes it easier to see the function-call chain, which can help reduce confusion when navigating around large C++ apps. For whatever it's worth, debugging OpenOffice.org with Memcheck is a bit easier when using this flag.

You don't have to do this, but doing so helps Valgrind produce more accurate and less confusing error reports. Chances are you're set up like this already, if you intended to debug your program with GNU gdb, or some other debugger.

This paragraph applies only if you plan to use Memcheck: On rare occasions, optimisation levels at `-O2` and above have been observed to generate code which fools Memcheck into wrongly reporting uninitialised value errors. We have looked in detail into fixing this, and unfortunately the result is that doing so would give a further significant slowdown in what is already a slow tool. So the best solution is to turn off optimisation altogether. Since this often makes things unmanagably slow, a plausible compromise is to use `-O`. This gets you the majority of the benefits of higher optimisation levels whilst keeping relatively small the chances of false complaints from Memcheck. All other tools (as far as we know) are unaffected by optimisation level.

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

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

2.3. The commentary

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

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

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

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

You can direct the commentary to three different places:

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

2. A less intrusive option is to write the commentary to a file, which you specify by `--log-file=filename`. Note carefully that the commentary is **not** written to the file you specify, but instead to one called `filename.pid12345`, if for example the pid of the traced process is 12345. This is helpful when valgrinding a whole tree of processes at once, since it means that each process writes to its own logfile, rather than the result being jumbled up in one big logfile. If `filename.pid12345` already exists, then it will name new files `filename.pid12345.1` and so on.

   If you want to specify precisely the file name to use, without the trailing `.pid12345`part, you can instead use `--log-file-exactly=filename`.

   You can also use the `--log-file-qualifier=<VAR>` option to specify the filename via the environment variable `$VAR`. This is rarely needed, but very useful in certain circumstances (eg. when running MPI programs).

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

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

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

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

   valgrind-listener accepts two command-line flags:

   - `-e` or `--exit-at-zero`: when the number of connected processes falls back to zero, exit. Without this, it will run forever, that is, until you send it Control-C.
   - `portnumber`: changes the port it listens on from the default (1500). The specified port must be in the range 1024 to 65535. The same restriction applies to port numbers specified by a `--log-socket=` to Valgrind itself.

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

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

2.4. Reporting of errors

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

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

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

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

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

Errors are reported before the associated operation actually happens. If you're using a tool (Memcheck, Addrcheck) which does address checking, and your program attempts to read from address zero, the tool will emit a message to this effect, and the program will then duly die with a segmentation fault.

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

The process of detecting duplicate errors is quite an expensive one and can become a significant performance overhead if your program generates huge quantities of errors. To avoid serious problems here, Valgrind will simply stop collecting errors after 300 different errors have been seen, or 30000 errors in total have been seen. In this situation you might as well stop your program and fix it, because Valgrind won't tell you anything else useful after this. Note that the 300/30000 limits apply after suppressed errors are removed. These limits are defined in `m_errormgr.c` and can be increased if necessary.

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

2.5. Suppressing errors

The error-checking tools detect numerous problems in the base libraries, such as the GNU C library, and the XFree86 client libraries, which come pre-installed on your GNU/Linux system. You can't easily fix these, but you don't want to see these errors (and yes, there are many!) So Valgrind reads a list of errors to suppress at startup. A default suppression file is cooked up by the `./configure` script when the system is built.

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

Note: By far the easiest way to add suppressions is to use the `--gen-suppressions=yes` flag described in Command-line flags for the Valgrind core [10].

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

If you use the `-v` flag, at the end of execution, Valgrind prints out one line for each used suppression, giving its name and the number of times it got used. Here's the suppressions used by a run of `valgrind --tool=memcheck ls -l`:

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

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

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

Each suppression has the following components:

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

  ```
  tool_name1,tool_name2:suppression_name
  ```

  Recall that Valgrind-2.0.X is a modular system, in which different instrumentation tools can observe your program whilst it is running. Since different tools detect different kinds of errors, it is necessary to say which tool(s) the suppression is meaningful to.

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

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

  ```
  core:PThread
  ```

- Next line: a small number of suppression types have extra information after the second line (eg. the `Param` suppression for Memcheck)
- Remaining lines: This is the calling context for the error -- the chain of function calls that led to it. There can be up to four of these lines.

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

  Important note: C++ function names must be **mangled**. If you are writing suppressions by hand, use the `--demangle=no` option to get the mangled names in your error messages.

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

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

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

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

(See Writing suppression files [35] for more details on the specifics of Memcheck's suppression kinds.)

Another example, again for the Memcheck tool:

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

Suppress any size 4 uninitialised-value error which occurs anywhere in `libX11.so.6.2`, when called from anywhere in the same library, when called from anywhere in `libXaw.so.7.0`. The inexact specification of locations is regrettable, but is about all you can hope for, given that the X11 libraries shipped with Red Hat 7.2 have had their symbol tables removed.

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

2.6. Command-line flags for the Valgrind core

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

You invoke Valgrind like this:

```
valgrind --tool=<emphasis>tool_name</emphasis> [valgrind-options]↩
your-prog [your-prog options]
```

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

2.6.1. Tool-selection option

The single most important option.

- `--tool=name`

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

2.6.2. Basic Options

These options work with all tools.

- `--help`

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

- `--help-debug`

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

- `--version`

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

- `-q --quiet`

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

- `-v --verbose`

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

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

  `--trace-children=yes`

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

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

  `--track-fds=yes`

  When enabled, Valgrind will print out a list of open file descriptors on exit. Along with each file descriptor, Valgrind prints out a stack backtrace of where the file was opened and any details relating to the file descriptor such as the file name or socket details.

- `--time-stamp=no` [default]

  `--time-stamp=yes`

  When enabled, Valgrind will precede each message with the current time and date.

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

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

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

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

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

  Just like `--log-file`, but the ".pid" suffix is not added. If you trace multiple processes with Valgrind when using this option the log file may get all messed up.

- `--log-file-qualifer=<VAR>`

  Specifies that Valgrind should send all of its messages to the file named by the environment variable `$VAR`. This is useful when running MPI programs.

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

  Specifies that Valgrind should send all of its messages to the specified port at the specified IP address. The port may be omitted, in which case port 1500 is used. If a connection cannot be made to the specified socket, Valgrind falls back to writing output to the standard error (stderr). This option is intended to be used in conjunction with the `valgrind-listener` program. For further details, see The commentary [6].

2.6.3. Error-related options

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

- `--xml=no` [default]

  `--xml=yes`

  When enabled, output will be in XML format. This is aimed at making life easier for tools that consume Valgrind's output as input, such as GUI front ends. Currently this option only works with Memcheck and Nulgrind.

- `--xml-user-comment=<string>` [default=""]

  Embeds an extra user comment string in the XML output. Only works with `--xml=yes` is specified; ignored otherwise.

- `--demangle=no`

  `--demangle=yes` [default]

  Disable/enable automatic demangling (decoding) of C++ names. Enabled by default. When enabled, Valgrind will attempt to translate encoded C++ procedure names back to something approaching the original. The demangler handles symbols mangled by g++ versions 2.X and 3.X.

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

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

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

  The maximum value for this is 50. Note that higher settings will make Valgrind run a bit more slowly and take a bit more memory, but can be useful when working with programs with deeply-nested call chains.

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

  `--error-limit=no`

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

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

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

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

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

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

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

  `--gen-suppressions=yes`

  `--gen-suppressions=all`

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

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

  If you choose to, Valgrind will print out a suppression for this error. You can then cut and paste it into a suppression file if you don't want to hear about the error in the future.

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

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

  Note that the suppressions printed are as specific as possible. You may want to common up similar ones, eg. by adding wildcards to function names. Also, sometimes two different errors are suppressed by the same suppression, in which case Valgrind will output the suppression more than once, but you only need to have one copy in your suppression file (but having more than one won't cause problems). Also, the suppression name is given as `<insert a suppression name here>`; the name doesn't really matter, it's only used with the `-v` option which prints out all used suppression records.

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

  `--db-attach=yes`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  In general, allocating large structures on the stack is a bad idea, because (1) you can easily run out of stack space, especially on systems with limited memory or which expect to support large numbers of threads each with a small stack, and (2) because the error checking performed by Memcheck is more effective for heap-allocated data than for stack-allocated data. If you have to use this flag, you may wish to consider rewriting your code to allocate on the heap rather than on the stack.

2.6.4. `malloc()`-related Options

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

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

  By default Valgrind's `malloc`, `realloc`, etc, return 8-byte aligned addresses. This is standard for most processors. Some programs might however assume that `malloc` et al return 16- or more aligned memory. The supplied value must be between 4 and 4096 inclusive, and must be a power of two.

2.6.5. Uncommon Options

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

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

  `--run-libc-freeres=no`

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

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

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

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

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

  - `lax-ioctls`

    Be very lax about ioctl handling; the only assumption is that the size is correct. Doesn't require the full buffer to be initialized when writing. Without this, using some device drivers with a large number of strange ioctl commands becomes very tiresome.

  - `ioctl-mmap`

    Some ioctl requests can mmap new memory into your process address space. If Valgrind doesn't know about these mappings, it could put new mappings over them, and/or complain bitterly when your program uses them. This option makes Valgrind scan the address space for new mappings after each unknown ioctl has finished. You may also need to run with `--pointercheck=no` if the ioctl decides to place the mapping out of the client's usual address space.

- `--pointercheck=yes` [default]

  `--pointercheck=no`

  This option make Valgrind generate a check on every memory reference to make sure it is within the client's part of the address space. This prevents stray writes from damaging Valgrind itself. On x86, this uses the CPU's segmentation machinery, and has almost no performance cost; there's almost never a reason to turn it off. On the other architectures this option is currently ignored as they don't have a cheap way of achieving the same functionality.

- `--show-emwarns=no` [default]

  `--show-emwarns=yes`

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

- `--smc-check=none`

  `--smc-check=stack` [default]

  `--smc-check=all`

  This option controls Valgrind's detection of self-modifying code. Valgrind can do no detection, detect self-modifying code on the stack, or detect self-modifying code anywhere. Note that the default option will catch the vast majority of cases, as far as we know. Running with `all` will slow Valgrind down greatly (but running with `none` will rarely speed things up, since very little code gets put on the stack for most programs).

2.6.6. Debugging Valgrind Options

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

2.6.7. Setting default Options

Note that Valgrind also reads options from three places:

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

These are processed in the given order, before the command-line options. Options processed later override those processed earlier; for example, options in `./.valgrindrc` will take precedence over those in `~/.valgrindrc`. The first two are particularly useful for setting the default tool to use.

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

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

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

2.7. The Client Request mechanism

Valgrind has a trapdoor mechanism via which the client program can pass all manner of requests and queries to Valgrind and the current tool. Internally, this is used extensively to make malloc, free, signals, threads, etc, work, although you don't see that.

For your convenience, a subset of these so-called client requests is provided to allow you to tell Valgrind facts about the behaviour of your program, and conversely to make queries. In particular, your program can tell Valgrind about changes in memory range permissions that Valgrind would not otherwise know about, and so allows clients to get Valgrind to do arbitrary custom checks.

Clients need to include a header file to make this work. Which header file depends on which client requests you use. Some client requests are handled by the core, and are defined in the header file `valgrind/valgrind.h`. Tool-specific header files are named after the tool, e.g. `valgrind/memcheck.h`. All header files can be found in the `include/valgrind` directory of wherever Valgrind was installed.

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

The code left in your binary has negligible performance impact. However, if you really wish to compile out the client requests, you can compile with `-DNVALGRIND` (analogous to `-DNDEBUG`'s effect on `assert()`).

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

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

`RUNNING_ON_VALGRIND`:

returns 1 if running on Valgrind, 0 if running on the real CPU. If you are running Valgrind under itself, it will return the number of layers of Valgrind emulation we're running under.

`VALGRIND_DISCARD_TRANSLATIONS`:

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

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

`VALGRIND_COUNT_ERRORS`:

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

`VALGRIND_MALLOCLIKE_BLOCK`:

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

`VALGRIND_FREELIKE_BLOCK`:

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

`VALGRIND_CREATE_MEMPOOL`:

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

`VALGRIND_DESTROY_MEMPOOL`:

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

`VALGRIND_MEMPOOL_ALLOC`:

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

`VALGRIND_MEMPOOL_FREE`:

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

`VALGRIND_NON_SIMD_CALL[0123]`:

executes a function of 0, 1, 2 or 3 args in the client program on the *real* CPU, not the virtual CPU that Valgrind normally runs code on. These are used in various ways internally to Valgrind. They might be useful to client programs.

Warning: Only use these if you *really* know what you are doing.

`VALGRIND_PRINTF(format, ...)`:

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

`VALGRIND_PRINTF_BACKTRACE(format, ...)`:

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

`VALGRIND_STACK_REGISTER(start, end)`:

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

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

`VALGRIND_STACK_DEREGISTER(id)`:

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

`VALGRIND_STACK_CHANGE(id, start, end)`:
: Change a previously registered stack. Informs Valgrind that the previously registerer stack with stack id `id` has changed it's start and end values. Use this if your user-level thread package implements stack growth.

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

2.8. Support for Threads

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

The main thing to point out is that although Valgrind works with the built-in threads system (eg. NPTL or LinuxThreads), it serialises execution so that only one thread is running at a time. This approach avoids the horrible implementation problems of implementing a truly multiprocessor version of Valgrind, but it does mean that threaded apps run only on one CPU, even if you have a multiprocessor machine.

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

2.9. Handling of Signals

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

If you're using signals in clever ways (for example, catching SIGSEGV, modifying page state and restarting the instruction), you're probably relying on precise exceptions. In this case, you will need to use `--single-step=yes`.

If your program dies as a result of a fatal core-dumping signal, Valgrind will generate its own core file (`vgcore.pidNNNNN`) containing your program's state. You may use this core file for post-mortem debugging with gdb or similar. (Note: it will not generate a core if your core dump size limit is 0.)

2.10. Building and Installing

We use the standard Unix `./configure`, `make`, `make install` mechanism, and we have attempted to ensure that it works on machines with kernel 2.4 or 2.6 and glibc 2.2.X, 2.3.X, 2.4.X.

There are two options (in addition to the usual `--prefix=` which affect how Valgrind is built:

- `--enable-pie`

  PIE stands for "position-independent executable". PIE allows Valgrind to place itself as high as possible in memory, giving your program as much address space as possible. It also allows Valgrind to run under itself. If PIE is disabled, Valgrind loads at a default address which is suitable for most systems. This is also useful for debugging Valgrind itself. It's not on by default because it caused problems for some people. Note that not all toolchaines support PIEs, you need fairly recent version of the compiler, linker, etc.

- `--enable-tls`

  TLS (Thread Local Storage) is a relatively new mechanism which requires compiler, linker and kernel support. Valgrind automatically test if TLS is supported and enable this option. Sometimes it cannot test for TLS, so this option allows you to override the automatic test.

The `configure` script tests the version of the X server currently indicated by the current `$DISPLAY`. This is a known bug. The intention was to detect the version of the current XFree86 client libraries, so that correct suppressions could be selected for them, but instead the test checks the server version. This is just plain wrong.

If you are building a binary package of Valgrind for distribution, please read `README_PACKAGERS` Readme Packagers [40]. It contains some important information.

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

2.11. If You Have Problems

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

See Limitations [21] for the known limitations of Valgrind, and for a list of programs which are known not to work on it.

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

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

Read the FAQ [http://www.valgrind.org/docs/faq/index.html] for more advice about common problems, crashes, etc.

2.12. Limitations

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

Valgrind will run x86/Linux ELF dynamically linked binaries, on a kernel 2.4.X or 2.6.X system, subject to the following constraints:

- On x86 and AMD64, there is no support for 3DNow! instructions. If the translator encounters these, Valgrind will generate a SIGILL when the instruction is executed.
- Atomic instruction sequences are not supported, which will affect any use of synchronization objects being shared between processes. They will appear to work, but fail sporadically.
- If your program does its own memory management, rather than using malloc/new/free/delete, it should still work, but Valgrind's error checking won't be so effective. If you describe your program's memory management scheme using "client requests" (Section 3.7 of this manual), Memcheck can do better. Nevertheless, using malloc/new and free/delete is still the best approach.

- Valgrind's signal simulation is not as robust as it could be. Basic POSIX-compliant sigaction and sigprocmask functionality is supplied, but it's conceivable that things could go badly awry if you do weird things with signals. Workaround: don't. Programs that do non-POSIX signal tricks are in any case inherently unportable, so should be avoided if possible.

- Machine instructions, and system calls, have been implemented on demand. So it's possible, although unlikely, that a program will fall over with a message to that effect. If this happens, please report ALL the details printed out, so we can try and implement the missing feature.

- Memory consumption of your program is majorly increased whilst running under Valgrind. This is due to the large amount of administrative information maintained behind the scenes. Another cause is that Valgrind dynamically translates the original executable. Translated, instrumented code is 14-16 times larger than the original (!) so you can easily end up with 30+ MB of translations when running (eg) a web browser.

- Valgrind can handle dynamically-generated code just fine. If you regenerate code over the top of old code (ie. at the same memory addresses), if the code is on the stack Valgrind will realise the code has changed, and work correctly. This is necessary to handle the trampolines GCC uses to implemented nested functions. If you regenerate code somewhere other than the stack, you will need to use the `--smc-check=all` flag, and Valgrind will run more slowly than normal.

- As of version 3.0.0, Valgrind has the following limitations in its implementation of floating point relative to the IEEE754 standard.

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

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

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

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

  For each exception, two courses of action are defined by 754: either (1) a user-defined exception handler may be called, or (2) a default action is defined, which "fixes things up" and allows the computation to proceed without throwing an exception.

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

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

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

- As of version 3.0.0, Valgrind has the following limitations in its implementation of x86/AMD64 SSE2 FP arithmetic.

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

Programs which are known not to work are:

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

Known platform-specific limitations, as of release 2.4.0:

- (none)

2.13. How It Works -- A Rough Overview

Some gory details, for those with a passion for gory details. You don’t need to read this section if all you want to do is use Valgrind. What follows is an outline of the machinery. It is out of date, as the JITter has been completey rewritten in version 3.0, and so it works quite differently. A more detailed (and even more out of date) description is to be found The design and implementation of Valgrind [5].

2.13.1. Getting started

Valgrind is compiled into two executables: `valgrind`, and `stage2`. `valgrind` is a statically-linked executable which loads at the normal address (0x8048000). `stage2` is a normal dynamically-linked executable; it is either linked to load at a high address (0xb8000000) or is a Position Independent Executable.

`Valgrind` (also known as `stage1`):

1. Decides where to load stage2.
2. Pads the address space with `mmap`, leaving holes only where stage2 should load.
3. Loads stage2 in the same manner as `execve()` would, but "manually".
4. Jumps to the start of stage2.

Once stage2 is loaded, it uses `dlopen()` to load the tool, unmaps all traces of stage1, initializes the client's state, and starts the synthetic CPU.

Each thread runs in its own kernel thread, and loops in `VG_(schedule)` as it runs. When the thread terminates, `VG_(schedule)` returns. Once all the threads have terminated, Valgrind as a whole exits.

Each thread also has two stacks. One is the client's stack, which is manipulated with the client's instructions. The other is Valgrind's internal stack, which is used by all Valgrind's code on behalf of that thread. It is important to not get them confused.

2.13.2. The translation/instrumentation engine

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

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

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

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

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

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

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

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

2.13.3. Tracking the Status of Memory

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

2.13.4. System calls

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

2.13.5. Signals

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

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

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

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

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

2.14. An Example Run

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

```
sewardj@phoenix:~/newmat10$
~/Valgrind-6/valgrind -v ./bogon
==25832== Valgrind 0.10, a memory error detector for x86 RedHat 7.1.
==25832== Copyright (C) 2000-2001, and GNU GPL'd, by Julian Seward.
==25832== Startup, with flags:
==25832== --suppressions=/home/sewardj/Valgrind/redhat71.supp
==25832== reading syms from /lib/ld-linux.so.2
==25832== reading syms from /lib/libc.so.6
==25832== reading syms from /mnt/pima/jrs/Inst/lib/libgcc_s.so.0
==25832== reading syms from /lib/libm.so.6
==25832== reading syms from /mnt/pima/jrs/Inst/lib/libstdc++.so.3
==25832== reading syms from /home/sewardj/Valgrind/valgrind.so
==25832== reading syms from /proc/self/exe
==25832== loaded 5950 symbols, 142333 line number locations
==25832==
==25832== Invalid read of size 4
==25832==    at 0x8048724: _ZN10BandMatrix6ReSizeEiii (bogon.cpp:45)
==25832==    by 0x80487AF: main (bogon.cpp:66)
==25832==  Address 0xBFFFF74C is not stack'd, malloc'd or free'd
==25832==
==25832== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==25832== malloc/free: in use at exit: 0 bytes in 0 blocks.
==25832== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==25832== For a detailed leak analysis, rerun with: --leak-check=yes
==25832==
==25832== exiting, did 1881 basic blocks, 0 misses.
==25832== 223 translations, 3626 bytes in, 56801 bytes out.
```

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

2.15. Warning Messages You Might See

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

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

  After 50 different errors have been shown, Valgrind becomes more conservative about collecting them. It then requires only the program counters in the top two stack frames to match when deciding whether or not two errors are really the same one. Prior to this point, the PCs in the top four frames are required to match. This hack has the effect of slowing down the appearance of new errors after the first 50. The 50 constant can be changed by recompiling Valgrind.

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

  After 300 different errors have been detected, Valgrind ignores any more. It seems unlikely that collecting even more different ones would be of practical help to anybody, and it avoids the danger that Valgrind spends more and more of its time comparing new errors against an ever-growing collection. As above, the 300 number is a compile-time constant.

- `Warning:  client switching stacks?`

  Valgrind spotted such a large change in the stack pointer, `%esp`, that it guesses the client is switching to a different stack. At this point it makes a kludgey guess where the base of the new stack is, and sets memory permissions accordingly. You may get many bogus error messages following this, if Valgrind guesses wrong. At the moment "large change" is defined as a change of more that 2000000 in the value of the `%esp` (stack pointer) register.

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

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

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

  Valgrind observed a call to one of the vast family of `ioctl` system calls, but did not modify its memory status info (because I have not yet got round to it). The call will still have gone through, but you may get spurious errors after this as a result of the non-update of the memory info.

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

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

3. Memcheck: a heavyweight memory checker

Table of Contents

3.1. Kinds of bugs that Memcheck can find 28
3.2. Command-line flags specific to Memcheck 28
3.3. Explanation of error messages from Memcheck 30
  3.3.1. Illegal read / Illegal write errors 30
  3.3.2. Use of uninitialised values 31
  3.3.3. Illegal frees 32
  3.3.4. When a block is freed with an inappropriate deallocation function 32
  3.3.5. Passing system call parameters with inadequate read/write permissions 33
  3.3.6. Overlapping source and destination blocks 34
  3.3.7. Memory leak detection 34
3.4. Writing suppression files 35
3.5. Details of Memcheck's checking machinery 36
  3.5.1. Valid-value (V) bits 36
  3.5.2. Valid-address (A) bits 37
  3.5.3. Putting it all together 38
3.6. Client Requests 39

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

3.1. Kinds of bugs that Memcheck can find

Memcheck is Valgrind's heavyweight memory checking tool. All reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted. As a result, Memcheck can detect the following problems:

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping `src` and `dst` pointers in `memcpy()` and related functions
- Some misuses of the POSIX pthreads API

3.2. Command-line flags specific to Memcheck

- `--leak-check=no`

  `--leak-check=summary` [default]

  `--leak-check=full`

  When enabled, search for memory leaks when the client program finishes. A memory leak means a malloc'd block, which has not yet been free'd, but to which no pointer can be found. Such a block can never be free'd by the program, since no pointer to it exists. If set to `summary`, it says how many leaks occurred. If set to `all`, it gives details of each individual leak.

- `--show-reachable=no` [default]

  `--show-reachable=yes`

  When disabled, the memory leak detector only shows blocks for which it cannot find a pointer to at all, or it can only find a pointer to the middle of. These blocks are prime candidates for memory leaks. When enabled, the leak detector also reports on blocks which it could find a pointer to. Your program could, at least in principle, have freed such blocks before exit. Contrast this to blocks for which no pointer, or only an interior pointer could be found: they are more likely to indicate memory leaks, because you do not actually have a pointer to the start of the block which you can hand to `free`, even if you wanted to.

- `--leak-resolution=low` [default]

  `--leak-resolution=med`

  `--leak-resolution=high`

  When doing leak checking, determines how willing Memcheck is to consider different backtraces to be the same. When set to `low`, the default, only the first two entries need match. When `med`, four entries have to match. When `high`, all entries need to match.

  For hardcore leak debugging, you probably want to use `--leak-resolution=high` together with `--num-callers=40` or some such large number. Note however that this can give an overwhelming amount of information, which is why the defaults are 4 callers and low-resolution matching.

  Note that the `--leak-resolution=` setting does not affect Memcheck's ability to find leaks. It only changes how the results are presented.

- `--freelist-vol=<number>` [default: 1000000]

  When the client program releases memory using free (in `C`) or delete (`C++`), that memory is not immediately made available for re-allocation. Instead it is marked inaccessible and placed in a queue of freed blocks. The purpose is to delay the point at which freed-up memory comes back into circulation. This increases the chance that Memcheck will be able to detect invalid accesses to blocks for some significant period of time after they have been freed.

  This flag specifies the maximum total size, in bytes, of the blocks in the queue. The default value is one million bytes. Increasing this increases the total amount of memory used by Memcheck but may detect invalid uses of freed blocks which would otherwise go undetected.

- `--workaround-gcc296-bugs=no` [default]

  `--workaround-gcc296-bugs=yes`

  When enabled, assume that reads and writes some small distance below the stack pointer are due to bugs in gcc 2.96, and does not report them. The "small distance" is 256 bytes by default. Note that gcc 2.96 is the default compiler on some popular Linux distributions (RedHat 7.X, Mandrake) and so you may well need to use this flag. Do not use it if you do not have to, as it can cause real errors to be overlooked. Another option is to use a gcc/g++ which does not generate accesses below the stack pointer. 2.95.3 seems to be a good choice in this respect.

  Unfortunately (27 Feb 02) it looks like g++ 3.0.4 has a similar bug, so you may need to issue this flag if you use 3.0.4. A while later (early Apr 02) this is confirmed as a scheduling bug in g++-3.0.4.

- `--partial-loads-ok=yes` [default]

  `--partial-loads-ok=no`

  Controls how Memcheck handles word (4-byte) loads from addresses for which some bytes are addressible and others are not. When `yes` (the default), such loads do not elicit an address error. Instead, the loaded V bytes corresponding to the illegal addresses indicate undefined, and those corresponding to legal addresses are loaded from shadow memory, as usual.

  When `no`, loads from partially invalid addresses are treated the same as loads from completely invalid addresses: an illegal-address error is issued, and the resulting V bytes indicate valid data.

- `--avoid-strlen-errors=no`

  `--avoid-strlen-errors=yes` [default]

  Enable or disable a heuristic for dealing with highly-optimized versions of strlen. These versions of strlen can cause spurious errors to be reported by Memcheck, so it's usually a good idea to leave this enabled.

3.3. Explanation of error messages from Memcheck

Despite considerable sophistication under the hood, Memcheck can only really detect two kinds of errors, use of illegal addresses, and use of undefined values. Nevertheless, this is enough to help you discover all sorts of memory-management nasties in your code. This section presents a quick summary of what error messages mean. The precise behaviour of the error-checking machinery is described in Details of Memcheck's checking machinery [36].

3.3.1. Illegal read / Illegal write errors

For example:

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40B07FF4: read_png_image__FP8QImageIO (kernel/qpngio.cpp:326)
   by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
 Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

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

Memcheck tries to establish what the illegal address might relate to, since that's often useful. So, if it points into a block of memory which has already been freed, you'll be informed of this, and also where the block was free'd at. Likewise, if it should turn out to be just off the end of a malloc'd block, a common result of off-by-one-errors in array subscripting, you'll be informed of this fact, and also where the block was malloc'd.

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

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

3.3.2. Use of uninitialised values

For example:

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

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

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

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

Sources of uninitialised data tend to be:

- Local variables in procedures which have not been initialised, as in the example above.

- The contents of malloc'd blocks, before you write something there. In C++, the new operator is a wrapper round malloc, so if you create an object with new, its fields will be uninitialised until you (or the constructor) fill them in, which is only Right and Proper.

3.3.3. Illegal frees

For example:

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

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

3.3.4. When a block is freed with an inappropriate deallocation function

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

```
Mismatched free() / delete / delete []
   at 0x40043249: free (vg_clientfuncs.c:171)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
 Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: __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)
```

The following was told to me be the KDE 3 developers. I didn't know any of it myself. They also implemented the check itself.

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

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

The worst thing is that on Linux apparently it doesn't matter if you do muddle these up, and it all seems to work ok, but the same program may then crash on a different platform, Solaris for example. So it's best to fix it properly. According to the KDE folks "it's amazing how many C++ programmers don't know this".

Pascal Massimino adds the following clarification: `delete[]` must be called associated with a `new[]` because the compiler stores the size of the array and the pointer-to-member to the destructor of the array's content just before the pointer actually returned. This implies a variable-sized overhead in what's returned by `new` or `new[]`.

3.3.5. Passing system call parameters with inadequate read/write permissions

Memcheck checks all parameters to system calls, i.e:

- It checks all the direct parameters themselves.
- Also, if a system call needs to read from a buffer provided by your program, Memcheck checks that the entire buffer is addressible and has valid data, ie, it is readable.
- Also, if the system call needs to write to a user-supplied buffer, Memcheck checks that the buffer is addressible.

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

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

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

You get these complaints ...

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

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

... because the program has (a) tried to write uninitialised junk from the malloc'd block to the standard output, and (b) passed an uninitialised value to `exit`. Note that the first error refers to the memory pointed to by `buf` (not `buf` itself), but the second error refers to the argument `error_code` itself.

3.3.6. Overlapping source and destination blocks

The following C library functions copy some data from one memory block to another (or something similar): `memcpy()`, `strcpy()`, `strncpy()`, `strcat()`, `strncat()`. The blocks pointed to by their `src` and `dst` pointers aren't allowed to overlap. Memcheck checks for this.

For example:

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

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

You might think that Memcheck is being overly pedantic reporting this in the case where `dst` is less than `src`. For example, the obvious way to implement `memcpy()` is by copying from the first byte to the last. However, the optimisation guides of some architectures recommend copying from the last byte down to the first. Also, some implementations of `memcpy()` zero `dst` before copying, because zeroing the destination's cache line(s) can improve performance.

The moral of the story is: if you want to write truly portable code, don't make any assumptions about the language implementation.

3.3.7. Memory leak detection

Memcheck keeps track of all memory blocks issued in response to calls to malloc/calloc/realloc/new. So when the program exits, it knows which blocks have not been freed.

If `--leak-check` is set appropriately, for each remaining block, Memcheck scans the entire address space of the process, looking for pointers to the block. Each block fits into one of the three following categories.

- Still reachable: A pointer to the start of the block is found. This usually indicates programming sloppiness; since the block is still pointed at, the programmer could, at least in principle, free'd it before program exit. Because these are very common and arguably not a problem, Memcheck won't report such blocks unless `--show-reachable=yes` is specified.
- Possibly lost, or "dubious": A pointer to the interior of the block is found. The pointer might originally have pointed to the start and have been moved along, or it might be entirely unrelated. Memcheck deems such a block as "dubious", because it's unclear whether or not a pointer to it still exists.
- Definitely lost, or "leaked": The worst outcome is that no pointer to the block can be found. The block is classified as "leaked", because the programmer could not possibly have freed it at program exit, since no pointer to it exists. This is likely a symptom of having lost the pointer at some earlier point in the program.

For each block mentioned, Memcheck will also tell you where the block was allocated. It cannot tell you how or why the pointer to a leaked block has been lost; you have to work that out for yourself. In general, you should attempt to ensure your programs do not have any leaked or dubious blocks at exit.

For example:

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:39)

88 (8 direct, 80 indirect) bytes in 1 blocks are definitely lost in loss↩
record 13 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:25)
```

The first message describes a simple case of a single 8 byte block that has been definitely lost. The second case mentions both "direct" and "indirect" leaks. The distinction is that a direct leak is a block which has no pointers to it. An indirect leak is a block which is only pointed to by other leaked blocks. Both kinds of leak are bad.

The precise area of memory in which Memcheck searches for pointers is: all naturally-aligned 4-byte words for which all A bits indicate addressibility and all V bits indicated that the stored value is actually valid.

3.4. Writing suppression files

The basic suppression format is described in Suppressing errors [8].

The suppression (2nd) line should have the form:

```
Memcheck:suppression_type
```

Or, since some of the suppressions are shared with Addrcheck:

```
Memcheck,Addrcheck:suppression_type
```

The Memcheck suppression types are as follows:

- `Value1`, `Value2`, `Value4`, `Value8`, `Value16`, meaning an uninitialised-value error when using a value of 1, 2, 4, 8 or 16 bytes.
- Or: `Cond` (or its old name, `Value0`), meaning use of an uninitialised CPU condition code.
- Or: `Addr1`, `Addr2`, `Addr4`, `Addr8`, `Addr16`, meaning an invalid address during a memory access of 1, 2, 4, 8 or 16 bytes respectively.
- Or: `Param`, meaning an invalid system call parameter error.
- Or: `Free`, meaning an invalid or mismatching free.

- Or: `Overlap`, meaning a `src` / `dst` overlap in `memcpy() or a similar function`.
- Or: `Leak`, meaning a memory leak.

The extra information line: for Param errors, is the name of the offending system call parameter. No other error kinds have this extra line.

The first line of the calling context: for Value and Addr errors, it is either the name of the function in which the error occurred, or, failing that, the full path of the .so file or executable containing the error location. For Free errors, is the name of the function doing the freeing (eg, `free`, `__builtin_vec_delete`, etc). For Overlap errors, is the name of the function with the overlapping arguments (eg. `memcpy()`, `strcpy()`, etc).

Lastly, there's the rest of the calling context.

3.5. Details of Memcheck's checking machinery

Read this section if you want to know, in detail, exactly what and how Memcheck is checking.

3.5.1. Valid-value (V) bits

It is simplest to think of Memcheck implementing a synthetic CPU which is identical to a real CPU, except for one crucial detail. Every bit (literally) of data processed, stored and handled by the real CPU has, in the synthetic CPU, an associated "valid-value" bit, which says whether or not the accompanying bit has a legitimate value. In the discussions which follow, this bit is referred to as the V (valid-value) bit.

Each byte in the system therefore has a 8 V bits which follow it wherever it goes. For example, when the CPU loads a word-size item (4 bytes) from memory, it also loads the corresponding 32 V bits from a bitmap which stores the V bits for the process' entire address space. If the CPU should later write the whole or some part of that value to memory at a different address, the relevant V bits will be stored back in the V-bit bitmap.

In short, each bit in the system has an associated V bit, which follows it around everywhere, even inside the CPU. Yes, all the CPU's registers (integer, floating point, and condition registers) have their own V bit vectors.

Copying values around does not cause Memcheck to check for, or report on, errors. However, when a value is used in a way which might conceivably affect the outcome of your program's computation, the associated V bits are immediately checked. If any of these indicate that the value is undefined, an error is reported.

Here's an (admittedly nonsensical) example:

```
int i, j;
int a[10], b[10];
for ( i = 0; i < 10; i++ ) {
  j = a[i];
  b[i] = j;
}
```

Memcheck emits no complaints about this, since it merely copies uninitialised values from `a[]` into `b[]`, and doesn't use them in any way. However, if the loop is changed to:

```
for ( i = 0; i < 10; i++ ) {
  j += a[i];
}
if ( j == 77 )
  printf("hello there\n");
```

then Valgrind will complain, at the `if`, that the condition depends on uninitialised values. Note that it **doesn’t** complain at the `j += a[i];`, since at that point the undefinedness is not "observable". It’s only when a decision has to be made as to whether or not to do the `printf` -- an observable action of your program -- that Memcheck complains.

Most low level operations, such as adds, cause Memcheck to use the V bits for the operands to calculate the V bits for the result. Even if the result is partially or wholly undefined, it does not complain.

Checks on definedness only occur in three places: when a value is used to generate a memory address, when control flow decision needs to be made, and when a system call is detected, Valgrind checks definedness of parameters as required.

If a check should detect undefinedness, an error message is issued. The resulting value is subsequently regarded as well-defined. To do otherwise would give long chains of error messages. In effect, we say that undefined values are non-infectious.

This sounds overcomplicated. Why not just check all reads from memory, and complain if an undefined value is loaded into a CPU register? Well, that doesn’t work well, because perfectly legitimate C programs routinely copy uninitialised values around in memory, and we don’t want endless complaints about that. Here’s the canonical example. Consider a struct like this:

```
struct S { int x; char c; };
struct S s1, s2;
s1.x = 42;
s1.c = ’z’;
s2 = s1;
```

The question to ask is: how large is `struct S`, in bytes? An `int` is 4 bytes and a `char` one byte, so perhaps a `struct S` occupies 5 bytes? Wrong. All (non-toy) compilers we know of will round the size of `struct S` up to a whole number of words, in this case 8 bytes. Not doing this forces compilers to generate truly appalling code for subscripting arrays of `struct S`’s.

So `s1` occupies 8 bytes, yet only 5 of them will be initialised. For the assignment `s2 = s1`, gcc generates code to copy all 8 bytes wholesale into `s2` without regard for their meaning. If Memcheck simply checked values as they came out of memory, it would yelp every time a structure assignment like this happened. So the more complicated semantics described above is necessary. This allows `gcc` to copy `s1` into `s2` any way it likes, and a warning will only be emitted if the uninitialised values are later used.

3.5.2. Valid-address (A) bits

Notice that the previous subsection describes how the validity of values is established and maintained without having to say whether the program does or does not have the right to access any particular memory location. We now consider the latter issue.

As described above, every bit in memory or in the CPU has an associated valid-value (V) bit. In addition, all bytes in memory, but not in the CPU, have an associated valid-address (A) bit. This indicates whether or not the program can legitimately read or write that location. It does not give any indication of the validity or the data at that location -- that's the job of the V bits -- only whether or not the location may be accessed.

Every time your program reads or writes memory, Memcheck checks the A bits associated with the address. If any of them indicate an invalid address, an error is emitted. Note that the reads and writes themselves do not change the A bits, only consult them.

So how do the A bits get set/cleared? Like this:

- When the program starts, all the global data areas are marked as accessible.
- When the program does malloc/new, the A bits for exactly the area allocated, and not a byte more, are marked as accessible. Upon freeing the area the A bits are changed to indicate inaccessibility.
- When the stack pointer register (`SP`) moves up or down, A bits are set. The rule is that the area from `SP` up to the base of the stack is marked as accessible, and below `SP` is inaccessible. (If that sounds illogical, bear in mind that the stack grows down, not up, on almost all Unix systems, including GNU/Linux.) Tracking `SP` like this has the useful side-effect that the section of stack used by a function for local variables etc is automatically marked accessible on function entry and inaccessible on exit.
- When doing system calls, A bits are changed appropriately. For example, mmap() magically makes files appear in the process's address space, so the A bits must be updated if mmap() succeeds.
- Optionally, your program can tell Valgrind about such changes explicitly, using the client request mechanism described above.

3.5.3. Putting it all together

Memcheck's checking machinery can be summarised as follows:

- Each byte in memory has 8 associated V (valid-value) bits, saying whether or not the byte has a defined value, and a single A (valid-address) bit, saying whether or not the program currently has the right to read/write that address.
- When memory is read or written, the relevant A bits are consulted. If they indicate an invalid address, Valgrind emits an Invalid read or Invalid write error.
- When memory is read into the CPU's registers, the relevant V bits are fetched from memory and stored in the simulated CPU. They are not consulted.
- When a register is written out to memory, the V bits for that register are written back to memory too.
- When values in CPU registers are used to generate a memory address, or to determine the outcome of a conditional branch, the V bits for those values are checked, and an error emitted if any of them are undefined.
- When values in CPU registers are used for any other purpose, Valgrind computes the V bits for the result, but does not check them.
- One the V bits for a value in the CPU have been checked, they are then set to indicate validity. This avoids long chains of errors.

- When values are loaded from memory, valgrind checks the A bits for that location and issues an illegal-address warning if needed. In that case, the V bits loaded are forced to indicate Valid, despite the location being invalid.

  This apparently strange choice reduces the amount of confusing information presented to the user. It avoids the unpleasant phenomenon in which memory is read from a place which is both unaddressible and contains invalid values, and, as a result, you get not only an invalid-address (read/write) error, but also a potentially large set of uninitialised-value errors, one for every time the value is used.

  There is a hazy boundary case to do with multi-byte loads from addresses which are partially valid and partially invalid. See details of the flag `--partial-loads-ok` for details.

Memcheck intercepts calls to malloc, calloc, realloc, valloc, memalign, free, new and delete. The behaviour you get is:

- malloc/new: the returned memory is marked as addressible but not having valid values. This means you have to write on it before you can read it.
- calloc: returned memory is marked both addressible and valid, since calloc() clears the area to zero.
- realloc: if the new size is larger than the old, the new section is addressible but invalid, as with malloc.
- If the new size is smaller, the dropped-off section is marked as unaddressible. You may only pass to realloc a pointer previously issued to you by malloc/calloc/realloc.
- free/delete: you may only pass to free a pointer previously issued to you by malloc/calloc/realloc, or the value NULL. Otherwise, Valgrind complains. If the pointer is indeed valid, Valgrind marks the entire area it points at as unaddressible, and places the block in the freed-blocks-queue. The aim is to defer as long as possible reallocation of this block. Until that happens, all attempts to access it will elicit an invalid-address error, as you would hope.

3.6. Client Requests

The following client requests are defined in `memcheck.h`. They also work for Addrcheck. See `memcheck.h` for exact details of their arguments.

- `VALGRIND_MAKE_NOACCESS`, `VALGRIND_MAKE_WRITABLE` and `VALGRIND_MAKE_READABLE`. These mark address ranges as completely inaccessible, accessible but containing undefined data, and accessible and containing defined data, respectively. Subsequent errors may have their faulting addresses described in terms of these blocks. Returns a "block handle". Returns zero when not run on Valgrind.
- `VALGRIND_DISCARD`: At some point you may want Valgrind to stop reporting errors in terms of the blocks defined by the previous three macros. To do this, the above macros return a small-integer "block handle". You can pass this block handle to `VALGRIND_DISCARD`. After doing so, Valgrind will no longer be able to relate addressing errors to the user-defined block associated with the handle. The permissions settings associated with the handle remain in place; this just affects how errors are reported, not whether they are reported. Returns 1 for an invalid handle and 0 for a valid handle (although passing invalid handles is harmless). Always returns 0 when not run on Valgrind.
- `VALGRIND_CHECK_WRITABLE` and `VALGRIND_CHECK_READABLE`: check immediately whether or not the given address range has the relevant property, and if not, print an error message. Also, for the convenience of the client, returns zero if the relevant property holds; otherwise, the returned value is the address of the first byte for which the property is not true. Always returns 0 when not run on Valgrind.

- `VALGRIND_CHECK_DEFINED`: a quick and easy way to find out whether Valgrind thinks a particular variable (lvalue, to be precise) is addressible and defined. Prints an error message if not. Returns no value.
- `VALGRIND_DO_LEAK_CHECK`: run the memory leak detector right now. Returns no value. I guess this could be used to incrementally check for leaks between arbitrary places in the program's execution. Warning: not properly tested!
- `VALGRIND_COUNT_LEAKS`: fills in the four arguments with the number of bytes of memory found by the previous leak check to be leaked, dubious, reachable and suppressed. Again, useful in test harness code, after calling `VALGRIND_DO_LEAK_CHECK`.
- `VALGRIND_GET_VBITS` and `VALGRIND_SET_VBITS`: allow you to get and set the V (validity) bits for an address range. You should probably only set V bits that you have got with `VALGRIND_GET_VBITS`. Only for those who really know what they are doing.

4. Addrcheck: a lightweight memory checker

Table of Contents

4.1. Kinds of bugs that Addrcheck can find 41

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

4.1. Kinds of bugs that Addrcheck can find

Addrcheck is a simplified version of the Memcheck tool described in Section 3. It is identical in every way to Memcheck, except for one important detail: it does not do the undefined-value checks that Memcheck does. This means Addrcheck is about twice as fast as Memcheck, and uses less memory. Addrcheck can detect the following errors:

- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping `src` and `dst` pointers in `memcpy()` and related functions
- Some misuses of the POSIX pthreads API

Rather than duplicate much of the Memcheck docs here (a.k.a. since I am a lazy b'stard), users of Addrcheck are advised to read Kinds of bugs that Memcheck can find [28]. Some important points:

- Addrcheck is exactly like Memcheck, except that all the value-definedness tracking machinery has been removed. Therefore, the Memcheck documentation which discusses definedess ("V-bits") is irrelevant. The stuff on addressibility ("A-bits") is still relevant.
- Addrcheck accepts the same command-line flags as Memcheck, with the exception of ... (to be filled in).
- Like Memcheck, Addrcheck will do memory leak checking (internally, the same code does leak checking for both tools). The only difference is how the two tools decide which memory locations to consider when searching for pointers to blocks. Memcheck will only consider 4-byte aligned locations which are validly addressible and which hold defined values. Addrcheck does not track definedness and so cannot apply the last, "defined value", criteria.

  The result is that Addrcheck's leak checker may "discover" pointers to blocks that Memcheck would not. So it is possible that Memcheck could (correctly) conclude that a block is leaked, yet Addrcheck would not conclude that.

  Whether or not this has any effect in practice is unknown. I suspect not, but that is mere speculation at this stage.

Addrcheck is, therefore, a fine-grained address checker. All it really does is check each memory reference to say whether or not that location may validly be addressed. Addrcheck has a memory overhead of one bit per byte of used address space. In contrast, Memcheck has an overhead of nine bits per byte.

Due to lazyness on the part of the implementor (Julian), error messages from Addrcheck do not distinguish reads from writes. So it will say, for example, "Invalid memory access of size 4", whereas Memcheck would have said whether the access is a read or a write. This could easily be remedied, if anyone is particularly bothered.

Addrcheck is quite pleasant to use. It's faster than Memcheck, and the lack of valid-value checks has another side effect: the errors it does report are relatively easy to track down, compared to the tedious and often confusing search sometimes needed to find the cause of uninitialised-value errors reported by Memcheck.

Because it is faster and lighter than Memcheck, our hope is that Addrcheck is more suitable for less-intrusive, larger scale testing than is viable with Memcheck. As of mid-November 2002, we have experimented with running the KDE-3.1 desktop on Addrcheck (the entire process tree, starting from `startkde`). Running on a 512MB, 1.7 GHz P4, the result is nearly usable. The ultimate aim is that is fast and unintrusive enough that (eg) KDE sessions may be unintrusively monitored for addressing errors whilst people do real work with their KDE desktop.

Addrcheck is a new experiment in the Valgrind world. We'd be interested to hear your feedback on it.

5. Cachegrind: a cache profiler

Table of Contents

5.1. Cache profiling 43
  5.1.1. Overview 43
  5.1.2. Cache simulation specifics 44
5.2. Profiling programs 45
  5.2.1. Output file 45
  5.2.2. Cachegrind options 46
  5.2.3. Annotating C/C++ programs 46
  5.2.4. Annotating assembler programs 51
5.3. cg_annotate options 51
  5.3.1. Warnings 52
  5.3.2. Things to watch out for 53
  5.3.3. Accuracy 54
  5.3.4. Todo 54

Detailed technical documentation on how Cachegrind works is available in How Cachegrind works [32]. If you only want to know how to **use** it, this is the page you need to read.

5.1. Cache profiling

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

Cachegrind is a tool for doing cache simulations and annotating your source line-by-line with the number of cache misses. In particular, it records:

- L1 instruction cache reads and misses;
- L1 data cache reads and read misses, writes and write misses;
- L2 unified cache reads and read misses, writes and writes misses.

On a modern machine, an L1 miss will typically cost around 10 cycles, and an L2 miss can cost as much as 200 cycles. Detailed cache profiling can be very useful for improving the performance of your program.

Also, since one instruction cache read is performed per instruction executed, you can find out how many instructions are executed per line, which can be useful for traditional profiling and test coverage.

Any feedback, bug-fixes, suggestions, etc, welcome.

5.1.1. Overview

First off, as for normal Valgrind use, you probably want to compile with debugging info (the `-g` flag). 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.

The two steps are:

1. Run your program with `valgrind --tool=cachegrind` in front of the normal command line invocation. When the program finishes, Cachegrind will print summary cache statistics. It also collects line-by-line information in a file `cachegrind.out.pid`, where `pid` is the program's process id.

   This step should be done every time you want to collect information about a new program, a changed program, or about the same program with different input.

2. Generate a function-by-function summary, and possibly annotate source files, using the supplied `cg_annotate` program. Source files to annotate can be specified manually, or manually on the command line, or "interesting" source files can be annotated automatically with the `--auto=yes` option. You can annotate C/C++ files or assembly language files equally easily.

   This step can be performed as many times as you like for each Step 2. You may want to do multiple annotations showing different information each time.

The steps are described in detail in the following sections.

5.1.2. Cache simulation specifics

Cachegrind uses a simulation for a machine with a split L1 cache and a unified L2 cache. This configuration is used for all (modern) x86-based machines we are aware of. Old Cyrix CPUs had a unified I and D L1 cache, but they are ancient history now.

The more specific characteristics of the simulation 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 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) = 2^N bytes
- Inclusive L2 cache: the L2 cache replicates all the entries of the L1 cache. This is standard on Pentium chips, but AMD Athlons use an exclusive L2 cache that only holds blocks evicted from L1. Ditto AMD Durons and most modern VIAs.

The cache configuration simulated (cache size, associativity and line size) is determined automagically using the CPUID instruction. If you have an old 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/L2) of the cache from the command line using the `--I1`, `--D1` and `--L2` 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 (eg. `inc` and `dec`) are counted as doing just a read, ie. 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 `vg_cachesim_I1.c`, `vg_cachesim_D1.c`, `vg_cachesim_L2.c` and `vg_cachesim_gen.c`. We'd be interested to hear from anyone who does.

5.2. Profiling programs

To gather cache profiling information about the program `ls -l`, invoke Cachegrind like this:

```
valgrind --tool=cachegrind ls -l
```

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

```
==31751== I   refs:      27,742,716
==31751== I1  misses:           276
==31751== L2  misses:           275
==31751== I1  miss rate:        0.0%
==31751== L2i miss rate:        0.0%
==31751==
==31751== D   refs:      15,430,290  (10,955,517 rd + 4,474,773 wr)
==31751== D1  misses:        41,185  (    21,905 rd +    19,280 wr)
==31751== L2  misses:        23,085  (     3,987 rd +    19,098 wr)
==31751== D1  miss rate:        0.2% (       0.1%   +       0.4%)
==31751== L2d miss rate:        0.1% (       0.0%   +       0.4%)
==31751==
==31751== L2 misses:         23,360  (     4,262 rd +    19,098 wr)
==31751== L2 miss rate:         0.0% (       0.0%   +       0.4%)
```

Cache accesses for instruction fetches are summarised first, giving the number of fetches made (this is the number of instructions executed, which can be useful to know in its own right), the number of I1 misses, and the number of L2 instruction (`L2i`) misses.

Cache accesses for data follow. The information is similar to that of the instruction fetches, except that the values are also shown split between reads and writes (note each row's `rd` and `wr` values add up to the row's total).

Combined instruction and data figures for the L2 cache follow that.

5.2.1. Output file

As well as printing summary information, Cachegrind also writes line-by-line cache profiling information to a file named `cachegrind.out.pid`. This file is human-readable, but is best interpreted by the accompanying program `cg_annotate`, described in the next section.

Things to note about the `cachegrind.out.pid` file:

- It is written every time Cachegrind is run, and will overwrite any existing `cachegrind.out.pid` in the current directory (but that won't happen very often because it takes some time for process ids to be recycled).
- It can be huge: `ls -l` generates a file of about 350KB. Browsing a few files and web pages with a Konqueror built with full debugging information generates a file of around 15 MB.

Note that older versions of Cachegrind used a log file named `cachegrind.out` (i.e. no `.pid` suffix). The suffix 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.

5.2.2. Cachegrind options

Cache-simulation specific options are:

```
--I1=<size>,<associativity>,<line_size>
--D1=<size>,<associativity>,<line_size>
--L2=<size>,<associativity>,<line_size>

[default: uses CPUID for automagic cache configuration]
```

Manually specifies the I1/D1/L2 cache configuration, where `size` and `line_size` are measured in bytes. The three items must be comma-separated, but with no spaces, eg:

```
valgrind --tool=cachegrind --I1=65535,2,64
```

You can specify one, two or three of the I1/D1/L2 caches. Any level not manually specified will be simulated using the configuration found in the normal way (via the CPUID instruction, or failing that, via defaults).

5.2.3. Annotating C/C++ programs

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

To get a function-by-function summary, run `cg_annotate --pid` in a directory containing a `cachegrind.out.pid` file. The *--pid* is required so that `cg_annotate` knows which log file to use when several are present.

The output looks like this:

```
------------------------------------------------------------------------------------

I1 cache:          65536 B, 64 B, 2-way associative
D1 cache:          65536 B, 64 B, 2-way associative
L2 cache:          262144 B, 64 B, 8-way associative
Command:           concord vg_to_ucode.c
Events recorded:     Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Events shown:        Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Event sort order:    Ir I1mr I2mr Dr D1mr D2mr Dw D1mw D2mw
Threshold:          99%
Chosen for annotation:
Auto-annotation:     on

------------------------------------------------------------------------------------

Ir       I1mr I2mr Dr        D1mr   D2mr  Dw       D1mw   D2mw
------------------------------------------------------------------------------------

27,742,716  276  275 10,955,517 21,905 3,987 4,474,773 19,280 19,098  PROGRAM↵
TOTALS

------------------------------------------------------------------------------------

Ir      I1mr I2mr Dr      D1mr  D2mr  Dw      D1mw   D2mw   file:function
------------------------------------------------------------------------------------

8,821,482   5   5 2,242,702 1,621   73 1,794,230     0     0  getc.c:_IO_getc
5,222,023   4   4 2,276,334   16   12   875,959     1     1  concord.c:get_word
2,649,248   2   2 1,344,810 7,326 1,385       .     .     .  vg_main.c:strcmp
2,521,927   2   2   591,215    0    0   179,398     0     0  concord.c:hash
2,242,740   2   2 1,046,612  568   22   448,548     0     0  ctype.c:tolower
1,496,937   4   4   630,874 9,000 1,400   279,388     0     0  concord.c:insert
  897,991  51  51   897,831   95   30       62     1     1  ???:???
  598,068   1   1   299,034    0    0   149,517     0     0↵
../sysdeps/generic/lockfile.c:__flockfile
  598,068   0   0   299,034    0    0   149,517     0     0↵
../sysdeps/generic/lockfile.c:__funlockfile
  598,024   4   4   213,580   35   16   149,506     0     0↵
vg_clientmalloc.c:malloc
  446,587   1   1   215,973 2,167  430   129,948 14,057 13,957↵
concord.c:add_existing
  341,760   2   2   128,160    0    0   128,160     0     0↵
vg_clientmalloc.c:vg_trap_here_WRAPPER
  320,782   4   4   150,711  276    0    56,027    53    53↵
concord.c:init_hash_table
  298,998   1   1   106,785    0    0    64,071     1     1  concord.c:create
  149,518   0   0   149,516    0    0        1     0     0↵
???:tolower@@GLIBC_2.0
  149,518   0   0   149,516    0    0        1     0     0  ???:fgetc@@GLIBC_2.0
   95,983   4   4    38,031    0    0    34,409  3,152  3,150↵
concord.c:new_word_node
   85,440   0   0    42,720    0    0    21,360     0     0↵
vg_clientmalloc.c:vg_bogus_epilogue
```

First up is a summary of the annotation options:

- I1 cache, D1 cache, L2 cache: cache configuration. So you know the configuration with which these results were obtained.
- Command: the command line invocation of the program under examination.
- Events recorded: event abbreviations are:
  - `Ir` : I cache reads (ie. instructions executed)
  - `I1mr`: I1 cache read misses
  - `I2mr`: L2 cache instruction read misses
  - `Dr` : D cache reads (ie. memory reads)
  - `D1mr`: D1 cache read misses
  - `D2mr`: L2 cache data read misses
  - `Dw` : D cache writes (ie. memory writes)
  - `D1mw`: D1 cache write misses
  - `D2mw`: L2 cache data write misses

  Note that D1 total accesses is given by `D1mr` + `D1mw`, and that L2 total accesses is given by `I2mr` + `D2mr` + `D2mw`.
- Events shown: the events shown (a subset of events gathered). This can be adjusted with the `--show` option.
- Event sort order: the sort order in which functions are shown. For example, in this case the functions are sorted from highest `Ir` counts to lowest. If two functions have identical `Ir` counts, they will then be sorted by `I1mr` counts, and so on. This order can be adjusted with the `--sort` option.

  Note that this dictates the order the functions appear. It is **not** the order in which the columns appear; that is dictated by the "events shown" line (and can be changed with the `--show` option).
- Threshold: `cg_annotate` by default omits functions that cause very low numbers of misses to avoid drowning you in information. In this case, cg_annotate shows summaries the functions that account for 99% of the `Ir` counts; `Ir` is chosen as the threshold event since it is the primary sort event. The threshold can be adjusted with the `--threshold` option.
- Chosen for annotation: names of files specified manually for annotation; in this case none.
- Auto-annotation: whether auto-annotation was requested via the `--auto=yes` option. In this case no.

Then follows summary statistics for the whole program. These are similar to the summary provided when running `valgrind --tool=cachegrind`.

Then follows function-by-function statistics. Each function is identified by a `file_name:function_name` pair. If a column contains only a dot it means the function never performs that event (eg. the third row shows that `strcmp()` contains no instructions that write to memory). The name `???` is used if the the file name and/or function name could not be determined from debugging information. If most of the entries have the form `???:???` the program probably wasn't compiled with `-g`. If any code was invalidated (either due to self-modifying code or unloading of shared objects) its counts are aggregated into a single cost centre written as `(discarded):(discarded)`.

It is worth noting that functions will come from three types of source files:

1. From the profiled program (`concord.c` in this example).
2. From libraries (eg. `getc.c`)
3. From Valgrind's implementation of some libc functions (eg. `vg_clientmalloc.c:malloc`). These are recognisable because the filename begins with `vg_`, and is probably one of `vg_main.c`, `vg_clientmalloc.c` or `vg_mylibc.c`.

There are two ways to annotate source files -- by choosing them manually, or with the `--auto=yes` option. To do it manually, just specify the filenames as arguments to `cg_annotate`. For example, the output from running `cg_annotate concord.c` for our example produces the same output as above followed by an annotated version of `concord.c`, a section of which looks like:

```
-----------------------------------------------------------------------------------

-- User-annotated source: concord.c
-----------------------------------------------------------------------------------

Ir      I1mr I2mr Dr     D1mr  D2mr  Dw     D1mw   D2mw

[snip]

      .   .   .     .    .    .     .    .    .  void init_hash_table(char↩
*file_name, Word_Node *table[])
      3   1   1     .    .    .     1    0    0  {
      .   .   .     .    .    .     .    .    .     FILE *file_ptr;
      .   .   .     .    .    .     .    .    .     Word_Info *data;
      1   0   0     .    .    .     1    1    1     int line = 1, i;
      .   .   .     .    .    .     .    .    .
      5   0   0     .    .    .     3    0    0     data = (Word_Info *)↩
create(sizeof(Word_Info));
      .   .   .     .    .    .     .    .    .
   4,991   0   0  1,995    0    0    998     0     0    for (i = 0; i <↩
TABLE_SIZE; i++)
   3,988   1   1  1,994    0    0    997    53    52       table[i] = NULL;
      .   .   .     .    .    .     .    .    .
      .   .   .     .    .    .     .    .    .     /* Open file, check it. */
      6   0   0     1    0    0     4    0    0     file_ptr =↩
fopen(file_name, "r");
      2   0   0     1    0    0     .    .    .     if (!(file_ptr)) {
      .   .   .     .    .    .     .    .    .        fprintf(stderr,↩
"Couldn’t open ’%s’.\n", file_name);
      1   1   1     .    .    .     .    .    .        exit(EXIT_FAILURE);
      .   .   .     .    .    .     .    .    .     }
      .   .   .     .    .    .     .    .    .
 165,062   1   1 73,360    0    0 91,700     0     0    while ((line =↩
get_word(data, line, file_ptr)) != EOF)
 146,712   0   0 73,356    0    0 73,356     0     0         ↩
insert(data->;word, data->line, table);
      .   .   .     .    .    .     .    .    .
      4   0   0     1    0    0     2    0    0     free(data);
      4   0   0     1    0    0     2    0    0     fclose(file_ptr);
      3   0   0     2    0    0     .    .    .  }
```

(Although column widths are automatically minimised, a wide terminal is clearly useful.)

Each source file is clearly marked (`User-annotated source`) as having been chosen manually for annotation. If the file was found in one of the directories specified with the `-I / --include` option, the directory and file are both given.

Each line is annotated with its event counts. Events not applicable for a line are represented by a ‘.’; this is useful for distinguishing between an event which cannot happen, and one which can but did not.

Sometimes only a small section of a source file is executed. To minimise uninteresting output, Valgrind only shows annotated lines and lines within a small distance of annotated lines. Gaps are marked with the line numbers so you know which part of a file the shown code comes from, eg:

```
(figures and code for line 704)
-- line 704 ----------------------------------------
-- line 878 ----------------------------------------
(figures and code for line 878)
```

The amount of context to show around annotated lines is controlled by the `--context` option.

To get automatic annotation, run `cg_annotate --auto=yes`. cg_annotate will automatically annotate every source file it can find that is mentioned in the function-by-function summary. Therefore, the files chosen for auto-annotation are affected by the `--sort` and `--threshold` options. Each source file is clearly marked (`Auto-annotated source`) as being chosen automatically. Any files that could not be found are mentioned at the end of the output, eg:

```
------------------------------------------------------------------
The following files chosen for auto-annotation could not be found:
------------------------------------------------------------------
  getc.c
  ctype.c
  ../sysdeps/generic/lockfile.c
```

This is quite common for library files, since libraries are usually compiled with debugging information, but the source files are often not present on a system. If a file is chosen for annotation **both** manually and automatically, it is marked as `User-annotated source`. Use the `-I / --include` option to tell Valgrind where to look for source files if the filenames found from the debugging information aren't specific enough.

Beware that cg_annotate can take some time to digest large `cachegrind.out.pid` files, e.g. 30 seconds or more. Also beware that auto-annotation can produce a lot of output if your program is large!

5.2.4. Annotating assembler programs

Valgrind can annotate assembler programs too, or annotate the assembler generated for your C program. Sometimes this is useful for understanding what is really happening when an interesting line of C code is translated into multiple instructions.

To do this, you just need to assemble your `.s` files with assembler-level debug information. gcc doesn't do this, but you can use the GNU assembler with the `--gstabs` option to generate object files with this information, eg:

```
as --gstabs foo.s
```

You can then profile and annotate source files in the same way as for C/C++ programs.

5.3. `cg_annotate` options

- `--pid`

  Indicates which `cachegrind.out.pid` file to read. Not actually an option -- it is required.

- `-h, --help`

  `-v, --version`

  Help and version, as usual.

- `--sort=A,B,C` [default: order in `cachegrind.out.pid`]

  Specifies the events upon which the sorting of the function-by-function entries will be based. Useful if you want to concentrate on eg. I cache misses (`--sort=I1mr,I2mr`), or D cache misses (`--sort=D1mr,D2mr`), or L2 misses (`--sort=D2mr,I2mr`).

- `--show=A,B,C` [default: all, using order in `cachegrind.out.pid`]

  Specifies which events to show (and the column order). Default is to use all present in the `cachegrind.out.pid` file (and use the order in the file).

- `--threshold=X` [default: 99%]

  Sets the threshold for the function-by-function summary. Functions are shown that account for more than X% of the primary sort event. If auto-annotating, also affects which files are annotated.

  Note: thresholds can be set for more than one of the events by appending any events for the `--sort` option with a colon and a number (no spaces, though). E.g. if you want to see the functions that cover 99% of L2 read misses and 99% of L2 write misses, use this option:

  ```
  --sort=D2mr:99,D2mw:99
  ```

- `--auto=no` [default]

  `--auto=yes`

  When enabled, automatically annotates every file that is mentioned in the function-by-function summary that can be found. Also gives a list of those that couldn't be found.

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

  Print N lines of context before and after each annotated line. Avoids printing large sections of source files that were not executed. Use a large number (eg. 10,000) to show all source lines.

- `-I=<dir>, --include=<dir>` [default: empty string]

  Adds a directory to the list in which to search for files. Multiple -I/--include options can be given to add multiple directories.

5.3.1. Warnings

There are a couple of situations in which `cg_annotate` issues warnings.

- If a source file is more recent than the `cachegrind.out.pid` file. This is because the information in `cachegrind.out.pid` is only recorded with line numbers, so if the line numbers change at all in the source (eg. 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, ie. shortening the source file while using an old `cachegrind.out.pid` file. If this happens, the figures for the bogus lines are printed anyway (clearly marked as bogus) in case they are important.

5.3.2. Things to watch out for

Some odd things that can occur during annotation:

- If annotating at the assembler level, you might see something like this:

  ```
  1   0   0 .   .   . .   .   .       leal -12(%ebp),%eax
  1   0   0 .   .   . 1   0   0       movl %eax,84(%ebx)
  2   0   0 0   0   0 1   0   0       movl $1,-20(%ebp)
  .   .   . .   .   . .   .   .       .align 4,0x90
  1   0   0 .   .   . .   .   .       movl $.LnrB,%eax
  1   0   0 .   .   . 1   0   0       movl %eax,-16(%ebp)
  ```

  How can the third instruction be executed twice when the others are executed only once? As it turns out, it isn't. Here's a dump of the executable, using `objdump -d`:

  ```
  8048f25:     8d 45 f4             lea   0xfffffff4(%ebp),%eax
  8048f28:     89 43 54             mov   %eax,0x54(%ebx)
  8048f2b:     c7 45 ec 01 00 00 00   movl  $0x1,0xffffffec(%ebp)
  8048f32:     89 f6                mov   %esi,%esi
  8048f34:     b8 08 8b 07 08         mov   $0x8078b08,%eax
  8048f39:     89 45 f0             mov   %eax,0xfffffff0(%ebp)
  ```

  Notice the extra `mov %esi,%esi` instruction. Where did this come from? The GNU assembler inserted it to serve as the two bytes of padding needed to align the `movl $.LnrB,%eax` instruction on a four-byte boundary, but pretended it didn't exist when adding debug information. Thus when Valgrind reads the debug info it thinks that the `movl $0x1,0xffffffec(%ebp)` instruction covers the address range 0x8048f2b--0x804833 by itself, and attributes the counts for the `mov %esi,%esi` to it.

- Inlined functions can cause strange results in the function-by-function summary. If a function `inline_me()` is defined in `foo.h` and inlined in the functions `f1()`, `f2()` and `f3()` in `bar.c`, there will not be a `foo.h:inline_me()` function entry. Instead, there will be separate function entries for each inlining site, ie. `foo.h:f1()`, `foo.h:f2()` and `foo.h:f3()`. To find the total counts for `foo.h:inline_me()`, add up the counts from each entry.

  The reason for this is that although the debug info output by gcc indicates the switch from `bar.c` to `foo.h`, it doesn't indicate the name of the function in `foo.h`, so Valgrind keeps using the old one.

- Sometimes, the same filename might be represented with a relative name and with an absolute name in different parts of the debug info, eg: `/home/user/proj/proj.h` and `../proj.h`. In this case, if you use auto-annotation, the file will be annotated twice with the counts split between the two.

- Files with more than 65,535 lines cause difficulties for the stabs debug info reader. This is because the line number in the `struct nlist` defined in `a.out.h` under Linux is only a 16-bit value. Valgrind can handle some files with more than 65,535 lines correctly by making some guesses to identify line number overflows. But some cases are beyond it, in which case you'll get a warning message explaining that annotations for the file might be incorrect.

- If you compile some files with `-g` and some without, some events that take place in a file without debug info could be attributed to the last line of a file with debug info (whichever one gets placed before the non-debug-info file in the executable).

This list looks long, but these cases should be fairly rare.

Note: `stabs` is not an easy format to read. If you come across bizarre annotations that look like might be caused by a bug in the stabs reader, please let us know.

5.3.3. Accuracy

Valgrind's cache profiling has a number of shortcomings:

- It doesn't account for kernel activity -- the effect of system calls on the cache contents is ignored.
- It doesn't account for other process activity (although this is probably desirable when considering a single program).
- It doesn't account for virtual-to-physical address mappings; hence the entire simulation is not a true representation of what's happening in the cache.
- It doesn't account for cache misses not visible at the instruction level, eg. those arising from TLB misses, or speculative execution.
- Valgrind's custom threads implementation will schedule threads differently to the standard one. This could warp the results for threaded programs.
- The instructions `bts`, `btr` and `btc` will incorrectly be counted as doing a data read if both the arguments are registers, eg:

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

  This should only happen rarely.
- 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 nothing is that results are very sensitive. Changing the size of the `valgrind.so` file, the size of the program being profiled, or even the length of its name can perturb the results. Variations will be small, but don't expect perfectly repeatable results if your program changes at all.

While these factors mean you shouldn't trust the results to be super-accurate, hopefully they should be close enough to be useful.

5.3.4. Todo

- Program start-up/shut-down calls a lot of functions that aren't interesting and just complicate the output. Would be nice to exclude these somehow.

6. Massif: a heap profiler

Table of Contents

6.1. Heap profiling 55
    6.1.1. Why Use a Heap Profiler? 56
6.2. Using Massif 56
    6.2.1. Overview 56
    6.2.2. Basic Results of Profiling 56
    6.2.3. Spacetime Graphs 57
6.3. Details of Heap Allocations 58
    6.3.1. Accuracy 60
6.4. Massif options 60

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

6.1. Heap profiling

Massif is a heap profiler, i.e. it measures how much heap memory programs use. In particular, it can give you information about:

- Heap blocks;
- Heap administration blocks;
- Stack sizes.

Heap profiling is useful to 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, avoid paging, and so on.
- 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.

6.1.1. Why Use a Heap Profiler?

Everybody knows how useful time profilers are for speeding up programs. They are particularly useful because people are notoriously bad at predicting where are the bottlenecks in their programs.

But the story is different for heap profilers. Some programming languages, particularly lazy functional languages like Haskell [http://www.haskell.org], have quite sophisticated heap profilers. But there are few tools as powerful for profiling C and C++ programs.

Why is this? Maybe it's because C and C++ programmers must think that they know where the memory is being allocated. After all, you can see all the calls to `malloc()` and `new` and `new[]`, right? But, in a big program, do you really know which heap allocations are being executed, how many times, and how large each allocation is? Can you give even a vague estimate of the memory footprint for your program? Do you know this for all the libraries your program uses? What about administration bytes required by the heap allocator to track heap blocks -- have you thought about them? What about the stack? If you are unsure about any of these things, maybe you should think about heap profiling.

Massif can tell you these things.

Or maybe it's because it's relatively easy to add basic heap profiling functionality into a program, to tell you how many bytes you have allocated for certain objects, or similar. But this information might only be simple like total counts for the whole program's execution. What about space usage at different points in the program's execution, for example? And reimplementing heap profiling code for each project is a pain.

Massif can save you this effort.

6.2. Using Massif

6.2.1. Overview

First off, as for normal Valgrind use, you probably want to compile with debugging info (the `-g` flag). But, as opposed to Memcheck, you probably **do** want to turn optimisation on, since you should profile your program as it will be normally run.

Then, run your program with `valgrind --tool=massif` in front of the normal command line invocation. When the program finishes, Massif will print summary space statistics. It also creates a graph representing the program's heap usage in a file called `massif.pid.ps`, which can be read by any PostScript viewer, such as Ghostview.

It also puts detailed information about heap consumption in a file `massif.pid.txt` (text format) or `massif.pid.html` (HTML format), where *pid* is the program's process id.

6.2.2. Basic Results of Profiling

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

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

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

```
==27519== Total spacetime:  2,258,106 ms.B
==27519== heap:           24.0%
==27519== heap admin:      2.2%
==27519== stack(s):       73.7%
```

All measurements are done in *spacetime*, i.e. space (in bytes) multiplied by time (in milliseconds). Note that because Massif slows a program down a lot, the actual spacetime figure is fairly meaningless; it's the relative values that are interesting.

Which entries you see in the breakdown depends on the command line options given. The above example measures all the possible parts of memory:

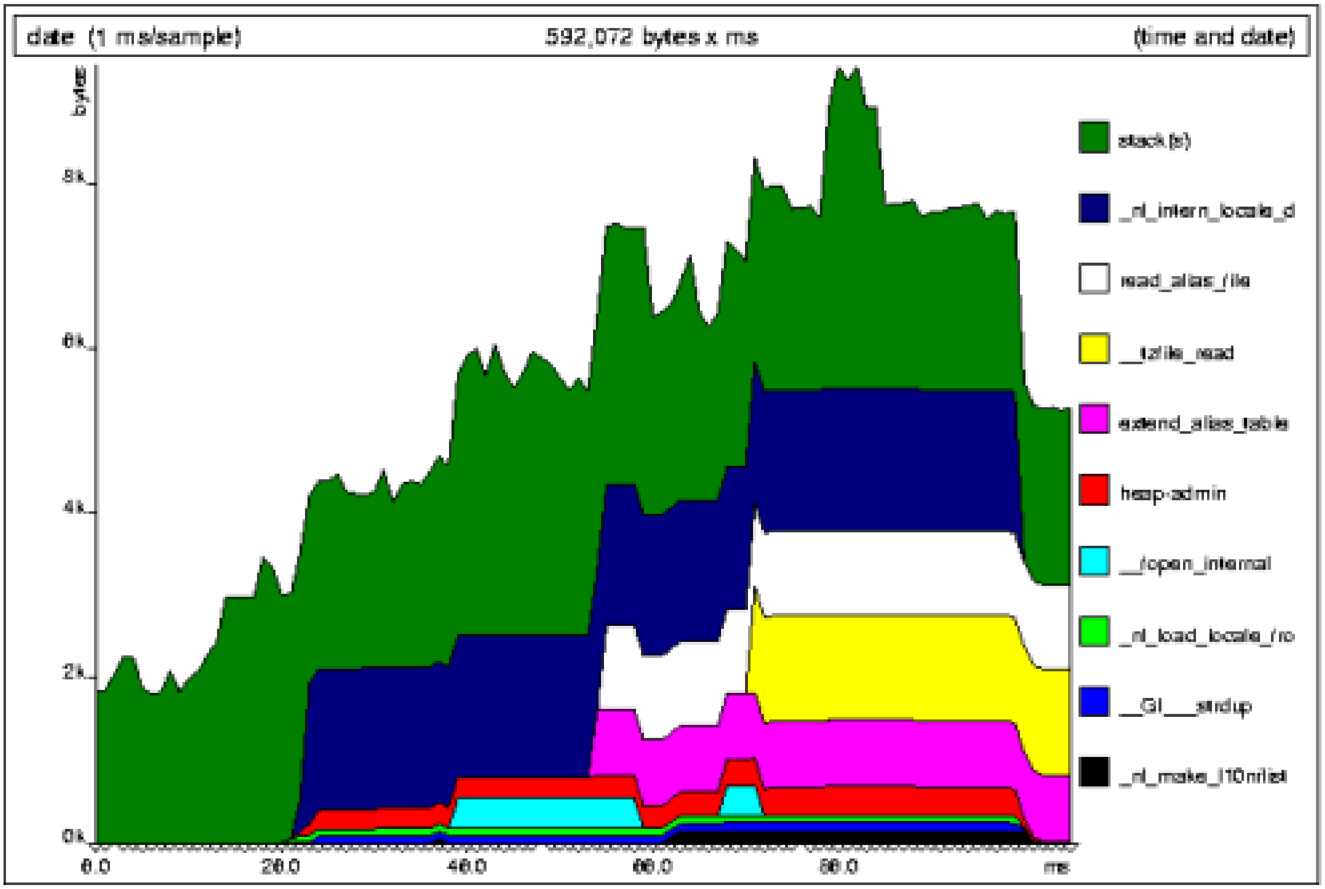
- Heap: number of words allocated on the heap, via `malloc()`, `new` and `new[]`.
- Heap admin: each heap block allocated requires some administration data, which lets the allocator track certain things about the block. It is easy to forget about this, and if your program allocates lots of small blocks, it can add up. This value is an estimate of the space required for this administration data.
- Stack(s): the spacetime used by the programs' stack(s). (Threaded programs can have multiple stacks.) This includes signal handler stacks.

6.2.3. Spacetime Graphs

As well as printing summary information, Massif also creates a file representing a spacetime graph, `massif.pid.hp`. It will produce a file called `massif.pid.ps`, which can be viewed in a PostScript viewer.

Massif uses a program called `hp2ps` to convert the raw data into the PostScript graph. It's distributed with Massif, but came originally from the Glasgow Haskell Compiler [http://haskell.cs.yale.edu/ghc/]. You shouldn't need to worry about this at all. However, if the graph creation fails for any reason, Massif will tell you, and will leave behind a file named `massif.pid.hp`, containing the raw heap profiling data.

Here's an example graph:



The graph is broken into several bands. Most bands represent a single line of your program that does some heap allocation; each such band represents all the allocations and deallocations done from that line. Up to twenty bands are shown; less significant allocation sites are merged into "other" and/or "OTHER" bands. The accompanying text/HTML file produced by Massif has more detail about these heap allocation bands. Then there are single bands for the stack(s) and heap admin bytes.

Note: it's the height of a band that's important. Don't let the ups and downs caused by other bands confuse you. For example, the `read_alias_file` band in the example has the same height all the time it's in existence.

The triangles on the x-axis show each point at which a memory census was taken. These aren't necessarily evenly spread; Massif only takes a census when memory is allocated or deallocated. The time on the x-axis is wallclock time, which is not ideal because you can get different graphs for different executions of the same program, due to random OS delays. But it's not too bad, and it becomes less of a problem the longer a program runs.

Massif takes censuses at an appropriate timescale; censuses take place less frequently as the program runs for longer. There is no point having more than 100-200 censuses on a single graph.

The graphs give a good overview of where your program's space use comes from, and how that varies over time. The accompanying text/HTML file gives a lot more information about heap use.

6.3. Details of Heap Allocations

The text/HTML file contains information to help interpret the heap bands of the graph. It also contains a lot of extra information about heap allocations that you don't see in the graph.

Here's part of the information that accompanies the above graph.

> == 0 ===========================
>
> Heap allocation functions accounted for 50.8% of measured spacetime
>
> Called from:
>
> - 22.1% [#b401767D1]: 0x401767D0: _nl_intern_locale_data (in /lib/i686/libc-2.3.2.so)
> - 8.6% [#b4017C394]: 0x4017C393: read_alias_file (in /lib/i686/libc-2.3.2.so)
> - ... ... *(several entries omitted)*
> - and 6 other insignificant places

The first part shows the total spacetime due to heap allocations, and the places in the program where most memory was allocated (Nb: if this program had been compiled with `-g`, actual line numbers would be given). These places are sorted, from most significant to least, and correspond to the bands seen in the graph. Insignificant sites (accounting for less than 0.5% of total spacetime) are omitted.

That alone can be useful, but often isn't enough. What if one of these functions was called from several different places in the program? Which one of these is responsible for most of the memory used? For `_nl_intern_locale_data()`, this question is answered by clicking on the 22.1% [#b401767D1] link, which takes us to the following part of the file:

> == 1 ===========================
>
> Context accounted for 22.1% [#a401767D1] of measured spacetime
>
> `0x401767D0: _nl_intern_locale_data (in /lib/i686/libc-2.3.2.so)`
>
> Called from:
>
> - 22.1% [#b40176F96]: 0x40176F95: _nl_load_locale_from_archive (in /lib/i686/libc-2.3.2.so)

At this level, we can see all the places from which `_nl_load_locale_from_archive()` was called such that it allocated memory at 0x401767D0. (We can click on the top 22.1% [#a40176F96] link to go back to the parent entry.) At this level, we have moved beyond the information presented in the graph. In this case, it is only called from one place. We can again follow the link for more detail, moving to the following part of the file.

```
== 2 ===========================
```

Context accounted for 22.1% [#a40176F96] of measured spacetime

```
0x401767D0:  _nl_intern_locale_data (in /lib/i686/libc-2.3.2.so)
0x40176F95:  _nl_load_locale_from_archive (in /lib/i686/libc-2.3.2.so)
```

Called from:

- 22.1%: 0x40176184: _nl_find_locale (in /lib/i686/libc-2.3.2.so)

In this way we can dig deeper into the call stack, to work out exactly what sequence of calls led to some memory being allocated. At this point, with a call depth of 3, the information runs out (thus the address of the child entry, 0x40176184, isn't a link). We could rerun the program with a greater `--depth` value if we wanted more information.

Sometimes you will get a code location like this:

```
30.8% : 0xFFFFFFFF: ???
```

The code address isn't really 0xFFFFFFFF -- that's impossible. This is what Massif does when it can't work out what the real code address is.

Massif produces this information in a plain text file by default, or HTML with the `--format=html` option. The plain text version obviously doesn't have the links, but a similar effect can be achieved by searching on the code addresses. (In Vim, the '*' and '#' searches are ideal for this.)

6.3.1. Accuracy

The information should be pretty accurate. Some approximations made might cause some allocation contexts to be attributed with less memory than they actually allocated, but the amounts should be miniscule.

The heap admin spacetime figure is an approximation, as described above. If anyone knows how to improve its accuracy, please let us know.

6.4. Massif options

Massif-specific options are:

- `--heap=no`

  `--heap=yes` [default]

  When enabled, profile heap usage in detail. Without it, the `massif.pid.txt` or `massif.pid.html` will be very short.

- `--heap-admin=n` [default: 8]

  The number of admin bytes per block to use. This can only be an estimate of the average, since it may vary. The allocator used by `glibc` requires somewhere between 4--15 bytes per block, depending on various factors. It also requires admin space for freed blocks, although Massif does not count this.

- `--stacks=no`

  `--stacks=yes` [default]

  When enabled, include stack(s) in the profile. Threaded programs can have multiple stacks.

- `--depth=n` [default: 3]

  Depth of call chains to present in the detailed heap information. Increasing it will give more information, but Massif will run the program more slowly, using more memory, and produce a bigger `.txt` / `.hp` file.

- `--alloc-fn=name`

  Specify a function that allocates memory. This is useful for functions that are wrappers to `malloc()`, which can fill up the context information uselessly (and give very uninformative bands on the graph). Functions specified will be ignored in contexts, i.e. treated as though they were `malloc()`. This option can be specified multiple times on the command line, to name multiple functions.

- `--format=text` [default]

  `--format=html`

  Produce the detailed heap information in text or HTML format. The file suffix used will be either `.txt` or `.html`.

7. Helgrind: a data-race detector

Helgrind is a Valgrind tool for detecting data races in C and C++ programs that use the Pthreads library.

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

It uses the Eraser algorithm described in:

Eraser: A Dynamic Data Race Detector for Multithreaded Programs
  Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro and Thomas Anderson
  ACM Transactions on Computer Systems, 15(4):391-411
  November 1997.

We also incorporate significant improvements from this paper:

Runtime Checking of Multithreaded Applications with Visual Threads
  Jerry J. Harrow, Jr.
  Proceedings of the 7th International SPIN Workshop on Model Checking of Software
  Stanford, California, USA
  August 2000
  LNCS 1885, pp331--342
  K. Havelund, J. Penix, and W. Visser, editors.

Basically what Helgrind does is to look for memory locations which are accessed by more than one thread. For each such location, Helgrind records which of the program's (pthread_mutex_)locks were held by the accessing thread at the time of the access. The hope is to discover that there is indeed at least one lock which is used by all threads to protect that location. If no such lock can be found, then there is (apparently) no consistent locking strategy being applied for that location, and so a possible data race might result.

Helgrind also allows for "thread segment lifetimes". If the execution of two threads cannot overlap -- for example, if your main thread waits on another thread with a `pthread_join()` operation -- they can both access the same variable without holding a lock.

There's a lot of other sophistication in Helgrind, aimed at reducing the number of false reports, and at producing useful error reports. We hope to have more documentation one day...

8. Nulgrind: the “null” tool

A tool that does not very much at all

Nulgrind is the minimal tool for Valgrind. It does no initialisation or finalisation, and adds no instrumentation to the program’s code. 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`).

9. Lackey: a very simple profiler

Lackey is a simple Valgrind tool that does some 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.

It measures three things:

1. The number of calls to `_dl_runtime_resolve()`, the function in glibc's dynamic linker that resolves function lookups into shared objects.
2. The number of UCode instructions (UCode is Valgrind's RISC-like intermediate language), x86 instructions, and basic blocks executed by the program, and some ratios between the three counts.
3. The number of conditional branches encountered and the proportion of those taken.

Valgrind FAQ

Valgrind Frequently Asked Questions

Valgrind Developers
[http://www.valgrind.org/www/developers.html]

Valgrind FAQ: Valgrind Frequently Asked Questions

Valgrind Developers [http://www.valgrind.org/www/developers.html]

1. Background

1.1. How do you pronounce "Valgrind"?

The "Val" as in the world "value". The "grind" is pronounced with a short ’i’ -- ie. "grinned" (rhymes with "tinned") rather than "grined" (rhymes with "find").

Don’t feel bad: almost everyone gets it wrong at first.

1.2. Where does the name "Valgrind" come from?

From Nordic mythology. Originally (before release) the project was named Heimdall, after the watchman of the Nordic gods. He could "see a hundred miles by day or night, hear the grass growing, see the wool growing on a sheep’s back" (etc). This would have been a great name, but it was already taken by a security package "Heimdal".

Keeping with the Nordic theme, Valgrind was chosen. Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard). Over this entrance there resides a wolf and over it there is the head of a boar and on it perches a huge eagle, whose eyes can see to the far regions of the nine worlds. Only those judged worthy by the guardians are allowed to pass through Valgrind. All others are refused entrance.

It’s not short for "value grinder", although that’s not a bad guess.

2. Compiling, installing and configuring

2.1. When I trying 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 www.mail-archive.com/bug-make@gnu.org/msg01658.html. Try upgrading to a more recent version of ’make’. Alternatively, we have heard that unsetting the CFLAGS environment variable avoids the problem.

3. Valgrind aborts unexpectedly

3.1. Programs run OK on Valgrind, but at exit produce a bunch of errors a bit like this:

```
==20755== Invalid read of size 4
==20755==    at 0x40281C8A: _nl_unload_locale (loadlocale.c:238)
==20755==    by 0x4028179D: free_mem (findlocale.c:257)
==20755==    by 0x402E0962: __libc_freeres (set-freeres.c:34)
==20755==    by 0x40048DCC: vgPlain___libc_freeres_wrapper (vg_clientfuncs.c:585)

==20755==    Address 0x40CC304C is 8 bytes inside a block of size 380 free'd
==20755==    at 0x400484C9: free (vg_clientfuncs.c:180)
==20755==    by 0x40281CBA: _nl_unload_locale (loadlocale.c:246)
==20755==    by 0x40281218: free_mem (setlocale.c:461)
==20755==    by 0x402E0962: __libc_freeres (set-freeres.c:34)
```

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.

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` flag. 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: vg_malloc2.c:442 (bszW_to_pszW): Assertion 'pszW >= 0' failed.
```

If Memcheck (the memory checker) shows any invalid reads, invalid writes and 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 like this. 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:

```
% disInstr: unhandled instruction bytes: 0x66 0xF 0x2E 0x5
```

Older versions did not support some x86 instructions, particularly SSE/SSE2 instructions. Try a newer Valgrind; we now support almost all instructions. If it still happens with newer versions, if the failing instruction is an SSE/SSE2 instruction, you might be able to recompile your program without it by using the flag `-march` to gcc. Either way, let us know and we'll try to fix it.

Another 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/Addrcheck may issue a warning just before this happens, but they might not if the jump happens to land in addressable memory.

4. Valgrind behaves unexpectedly

4.1. My threaded server process runs unbelievably slowly on Valgrind. So slowly, in fact, that at first I thought it had completely locked up.

We are not completely sure about this, but one possibility is that laptops with power management fool Valgrind's timekeeping mechanism, which is (somewhat in error) based on the x86 RDTSC instruction. A "fix" which is claimed to work is to run some other cpu-intensive process at the same time, so that the laptop's power-management clock-slowing does not kick in. We would be interested in hearing more feedback on this.

Another possible cause is that versions prior to 1.9.6 did not support threading on glibc 2.3.X systems well. Hopefully the situation is much improved with 1.9.6 and later versions.

4.2. 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 is 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++/ext/howto.html#3 [http://gcc.gnu.org/onlinedocs/libstdc++/ext/howto.html#3] if you absolutely want to do that. But beware:

1. there are currently changes underway for gcc which are not totally reflected in the docs right now ("now" == 26 Apr 03)
2. allocators belong to the more messy parts of the STL and people went to great lengths to make it portable across platforms. Chances are good that your solution will work on your platform, but not on others.

4.3. 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. The workaround here 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)
```

4.4. 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'.

5. Memcheck doesn't find my bug

5.1. I try running "valgrind --tool=memcheck my_program" and get Valgrind's startup message, but I don't get 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 flag `--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: Directing output to file [http://www.valgrind.org/docs/bookset/manual-core.out2file.html] for more details.

Second, if your program is statically linked, most Valgrind tools won't work as well, because they won't be able to replace certain functions, such as malloc(), with their own versions. A key indicator of this is if Memcheck says:

```
No malloc'd blocks -- no leaks are possible
```

when you know your program calls malloc(). The workaround is to avoid statically linking your program.

5.2. 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 static 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.

6. Miscellaneous

6.1. 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_`.

6.2. With Memcheck/Addrcheck’s memory leak detector, what’s the difference between "definitely lost", "possibly lost", "still reachable", and "suppressed"?

The details are in the Manual: Memory leak detection [http://www.valgrind.org/docs/bookset/mc-manual.leaks.html].

In short:

- "definitely lost" means your program is leaking memory -- fix it!
- "possibly lost" means your program is probably leaking memory, unless you’re doing funny things with pointers.
- "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.

7. How To Get Further Assistance

Please read all of this section before posting.

If you think an answer is incomplete or inaccurate, please e-mail valgrind@valgrind.org [mailto:valgrind@valgrind.org].

Read the appropriate section(s) of the Manual(s): Valgrind Documentation [http://www.valgrind.org/docs/].

Read the Distribution Documents [http://www.valgrind.org/docs/].

Search [http://search.gmane.org] the valgrind-users [http://news.gmane.org/gmane.comp.debugging.valgrind] mailing list archives, using the group name `gmane.comp.debugging.valgrind`.

Only when you have tried all of these things and are still stuck, should you post to the valgrind-users mailing list [http://lists.sourceforge.net/lists/listinfo/valgrind-users]. In which case, please read the following carefully. Making a complete posting will greatly increase the chances that an expert or fellow user reading it will have enough information and motivation to reply.

Make sure you give full details of the problem, including the full output of `valgrind -v`, if applicable. Also which Linux distribution you're using (Red Hat, Debian, etc) and its version number.

You are in little danger of making your posting too long unless you include large chunks of valgrind's (unsuppressed) output, so err on the side of giving too much information.

Clearly written subject lines and message bodies are appreciated, too.

Finally, 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

Valgrind Technical Documentation

Table of Contents

1. The Design and Implementation of Valgrind 5
  - 1.1. Introduction 5
    - 1.1.1. History 6
    - 1.1.2. Design overview 6
    - 1.1.3. Design decisions 7
    - 1.1.4. Correctness 8
    - 1.1.5. Current limitations 10
  - 1.2. The instrumenting JITter 10
    - 1.2.1. Run-time storage, and the use of host registers 10
    - 1.2.2. Startup, shutdown, and system calls 12
    - 1.2.3. Introduction to UCode 12
    - 1.2.4. UCode operand tags: type Tag 13
    - 1.2.5. UCode instructions: type UInstr 14
    - 1.2.6. Translation into UCode 17
    - 1.2.7. UCode optimisation 19
    - 1.2.8. UCode instrumentation 20
    - 1.2.9. UCode post-instrumentation cleanup 23
    - 1.2.10. Translation from UCode 25
    - 1.2.11. Top-level dispatch loop 26
    - 1.2.12. Lazy updates of the simulated program counter 26
    - 1.2.13. Signals 26
    - 1.2.14. To be written 26
  - 1.3. Extensions 27
    - 1.3.1. Bugs 27
    - 1.3.2. Threads 27
    - 1.3.3. Verification suite 27
    - 1.3.4. Porting to other platforms 27
  - 1.4. Easy stuff which ought to be done 28
    - 1.4.1. MMX Instructions 28
    - 1.4.2. Fix stabs-info reader 28
    - 1.4.3. BT/BTC/BTS/BTR 28
    - 1.4.4. Using PREFETCH Instructions 28
    - 1.4.5. User-defined Permission Ranges 29

2. How Cachegrind works 32
  - 2.1. Cache profiling 32
  - 2.2. Cost centres 32
  - 2.3. Storing cost-centres 34
  - 2.4. Instrumentation 35
  - 2.5. Handling basic block retranslations 37
  - 2.6. The cache simulation 38
  - 2.7. Output 38
  - 2.8. Summary of performance features 39
  - 2.9. Annotation 39
  - 2.10. Similar work, extensions 39

3. Writing a New Valgrind Tool 40
  - 3.1. Introduction 40
    - 3.1.1. Supervised Execution 40
    - 3.1.2. Tools 40
    - 3.1.3. Execution Spaces 41
  - 3.2. Writing a Tool 41
    - 3.2.1. Why write a tool? 41
    - 3.2.2. Suggested tools 42

- 3.2.3. How tools work 43
- 3.2.4. Getting the code 43
- 3.2.5. Getting started 43
- 3.2.6. Writing the code 45
- 3.2.7. Initialisation 45
- 3.2.8. Instrumentation 45
- 3.2.9. Finalisation 46
- 3.2.10. Other Important Information 46
- 3.2.11. Words of Advice 46

3.3. Advanced Topics 47
- 3.3.1. Suppressions 47
- 3.3.2. Documentation 47
- 3.3.3. Regression Tests 49
- 3.3.4. Profiling 50
- 3.3.5. Other Makefile Hackery 50
- 3.3.6. Core/tool Interface Versions 50

3.4. Final Words 50

1. The Design and Implementation of Valgrind

Detailed technical notes for hackers, maintainers and the overly-curious

Table of Contents

1.1. Introduction . . . 5
  1.1.1. History . . . 6
  1.1.2. Design overview . . . 6
  1.1.3. Design decisions . . . 7
  1.1.4. Correctness . . . 8
  1.1.5. Current limitations . . . 10
1.2. The instrumenting JITter . . . 10
  1.2.1. Run-time storage, and the use of host registers . . . 10
  1.2.2. Startup, shutdown, and system calls . . . 12
  1.2.3. Introduction to UCode . . . 12
  1.2.4. UCode operand tags: type Tag . . . 13
  1.2.5. UCode instructions: type UInstr . . . 14
  1.2.6. Translation into UCode . . . 17
  1.2.7. UCode optimisation . . . 19
  1.2.8. UCode instrumentation . . . 20
  1.2.9. UCode post-instrumentation cleanup . . . 23
  1.2.10. Translation from UCode . . . 25
  1.2.11. Top-level dispatch loop . . . 26
  1.2.12. Lazy updates of the simulated program counter . . . 26
  1.2.13. Signals . . . 26
  1.2.14. To be written . . . 26
1.3. Extensions . . . 27
  1.3.1. Bugs . . . 27
  1.3.2. Threads . . . 27
  1.3.3. Verification suite . . . 27
  1.3.4. Porting to other platforms . . . 27
1.4. Easy stuff which ought to be done . . . 28
  1.4.1. MMX Instructions . . . 28
  1.4.2. Fix stabs-info reader . . . 28
  1.4.3. BT/BTC/BTS/BTR . . . 28
  1.4.4. Using PREFETCH Instructions . . . 28
  1.4.5. User-defined Permission Ranges . . . 29

1.1. Introduction

This document contains a detailed, highly-technical description of the internals of Valgrind. This is not the user manual; if you are an end-user of Valgrind, you do not want to read this. Conversely, if you really are a hacker-type and want to know how it works, I assume that you have read the user manual thoroughly.

You may need to read this document several times, and carefully. Some important things, I only say once.

[Note: this document is now very old, and a lot of its contents are out of date, and misleading.]

1.1.1. History

Valgrind came into public view in late Feb 2002. However, it has been under contemplation for a very long time, perhaps seriously for about five years. Somewhat over two years ago, I started working on the x86 code generator for the Glasgow Haskell Compiler (http://www.haskell.org/ghc), gaining familiarity with x86 internals on the way. I then did Cacheprof, gaining further x86 experience. Some time around Feb 2000 I started experimenting with a user-space x86 interpreter for x86-Linux. This worked, but it was clear that a JIT-based scheme would be necessary to give reasonable performance for Valgrind. Design work for the JITter started in earnest in Oct 2000, and by early 2001 I had an x86-to-x86 dynamic translator which could run quite large programs. This translator was in a sense pointless, since it did not do any instrumentation or checking.

Most of the rest of 2001 was taken up designing and implementing the instrumentation scheme. The main difficulty, which consumed a lot of effort, was to design a scheme which did not generate large numbers of false uninitialised-value warnings. By late 2001 a satisfactory scheme had been arrived at, and I started to test it on ever-larger programs, with an eventual eye to making it work well enough so that it was helpful to folks debugging the upcoming version 3 of KDE. I’ve used KDE since before version 1.0, and wanted to Valgrind to be an indirect contribution to the KDE 3 development effort. At the start of Feb 02 the kde-core-devel crew started using it, and gave a huge amount of helpful feedback and patches in the space of three weeks. Snapshot 20020306 is the result.

In the best Unix tradition, or perhaps in the spirit of Fred Brooks’ depressing-but-completely-accurate epitaph "build one to throw away; you will anyway", much of Valgrind is a second or third rendition of the initial idea. The instrumentation machinery (`vg_translate.c`, `vg_memory.c`) and core CPU simulation (`vg_to_ucode.c`, `vg_from_ucode.c`) have had three redesigns and rewrites; the register allocator, low-level memory manager (`vg_malloc2.c`) and symbol table reader (`vg_symtab2.c`) are on the second rewrite. In a sense, this document serves to record some of the knowledge gained as a result.

1.1.2. Design overview

Valgrind is compiled into a Linux shared object, `valgrind.so`, and also a dummy one, `valgrinq.so`, of which more later. The `valgrind` shell script adds `valgrind.so` to the `LD_PRELOAD` list of extra libraries to be loaded with any dynamically linked library. This is a standard trick, one which I assume the `LD_PRELOAD` mechanism was developed to support.

`valgrind.so` is linked with the `-z initfirst` flag, which requests that its initialisation code is run before that of any other object in the executable image. When this happens, valgrind gains control. The real CPU becomes "trapped" in `valgrind.so` and the translations it generates. The synthetic CPU provided by Valgrind does, however, return from this initialisation function. So the normal startup actions, orchestrated by the dynamic linker `ld.so`, continue as usual, except on the synthetic CPU, not the real one. Eventually `main` is run and returns, and then the finalisation code of the shared objects is run, presumably in inverse order to which they were initialised. Remember, this is still all happening on the simulated CPU. Eventually `valgrind.so`’s own finalisation code is called. It spots this event, shuts down the simulated CPU, prints any error summaries and/or does leak detection, and returns from the initialisation code on the real CPU. At this point, in effect the real and synthetic CPUs have merged back into one, Valgrind has lost control of the program, and the program finally `exit()s` back to the kernel in the usual way.

The normal course of activity, once Valgrind has started up, is as follows. Valgrind never runs any part of your program (usually referred to as the "client"), not a single byte of it, directly. Instead it uses function `VG_(translate)` to translate basic blocks (BBs, straight-line sequences of code) into instrumented translations, and those are run instead. The translations are stored in the translation cache (TC), `vg_tc`, with the translation table (TT), `vg_tt` supplying the original-to-translation code address mapping. Auxiliary array `VG_(tt_fast)` is used as a direct-map cache for fast lookups in TT; it usually achieves a hit rate of around 98% and facilitates an orig-to-trans lookup in 4 x86 insns, which is not bad.

Function `VG_(dispatch)` in `vg_dispatch.S` is the heart of the JIT dispatcher. Once a translated code address has been found, it is executed simply by an x86 `call` to the translation. At the end of the translation, the next original code addr is loaded into `%eax`, and the translation then does a `ret`, taking it back to the dispatch loop, with, interestingly, zero branch mispredictions. The address requested in `%eax` is looked up first in `VG_(tt_fast)`, and, if not found, by calling C helper `VG_(search_transtab)`. If there is still no translation available, `VG_(dispatch)` exits back to the top-level C dispatcher `VG_(toploop)`, which arranges for `VG_(translate)` to make a new translation. All fairly unsurprising, really. There are various complexities described below.

The translator, orchestrated by `VG_(translate)`, is complicated but entirely self-contained. It is described in great detail in subsequent sections. Translations are stored in TC, with TT tracking administrative information. The translations are subject to an approximate LRU-based management scheme. With the current settings, the TC can hold at most about 15MB of translations, and LRU passes prune it to about 13.5MB. Given that the orig-to-translation expansion ratio is about 13:1 to 14:1, this means TC holds translations for more or less a megabyte of original code, which generally comes to about 70000 basic blocks for C++ compiled with optimisation on. Generating new translations is expensive, so it is worth having a large TC to minimise the (capacity) miss rate.

The dispatcher, `VG_(dispatch)`, receives hints from the translations which allow it to cheaply spot all control transfers corresponding to x86 `call` and `ret` instructions. It has to do this in order to spot some special events:

- Calls to `VG_(shutdown)`. This is Valgrind's cue to exit. NOTE: actually this is done a different way; it should be cleaned up.
- Returns of system call handlers, to the return address `VG_(signalreturn_bogusRA)`. The signal simulator needs to know when a signal handler is returning, so we spot jumps (returns) to this address.
- Calls to `vg_trap_here`. All `malloc`, `free`, etc calls that the client program makes are eventually routed to a call to `vg_trap_here`, and Valgrind does its own special thing with these calls. In effect this provides a trapdoor, by which Valgrind can intercept certain calls on the simulated CPU, run the call as it sees fit itself (on the real CPU), and return the result to the simulated CPU, quite transparently to the client program.

Valgrind intercepts the client's `malloc`, `free`, etc, calls, so that it can store additional information. Each block `malloc`'d by the client gives rise to a shadow block in which Valgrind stores the call stack at the time of the `malloc` call. When the client calls `free`, Valgrind tries to find the shadow block corresponding to the address passed to `free`, and emits an error message if none can be found. If it is found, the block is placed on the freed blocks queue `vg_freed_list`, it is marked as inaccessible, and its shadow block now records the call stack at the time of the `free` call. Keeping `free`'d blocks in this queue allows Valgrind to spot all (presumably invalid) accesses to them. However, once the volume of blocks in the free queue exceeds `VG_(clo_freelist_vol)`, blocks are finally removed from the queue.

Keeping track of `A` and `V` bits (note: if you don't know what these are, you haven't read the user guide carefully enough) for memory is done in `vg_memory.c`. This implements a sparse array structure which covers the entire 4G address space in a way which is reasonably fast and reasonably space efficient. The 4G address space is divided up into 64K sections, each covering 64Kb of address space. Given a 32-bit address, the top 16 bits are used to select one of the 65536 entries in `VG_(primary_map)`. The resulting "secondary" (`SecMap`) holds A and V bits for the 64k of address space chunk corresponding to the lower 16 bits of the address.

1.1.3. Design decisions

Some design decisions were motivated by the need to make Valgrind debuggable. Imagine you are writing a CPU simulator. It works fairly well. However, you run some large program, like Netscape, and after tens of millions of instructions, it crashes. How can you figure out where in your simulator the bug is?

Valgrind's answer is: cheat. Valgrind is designed so that it is possible to switch back to running the client program on the real CPU at any point. Using the `--stop-after=` flag, you can ask Valgrind to run just some number of basic blocks, and then run the rest of the way on the real CPU. If you are searching for a bug in the simulated CPU, you can use this to do a binary search, which quickly leads you to the specific basic block which is causing the problem.

This is all very handy. It does constrain the design in certain unimportant ways. Firstly, the layout of memory, when viewed from the client's point of view, must be identical regardless of whether it is running on the real or simulated CPU. This means that Valgrind can't do pointer swizzling -- well, no great loss -- and it can't run on the same stack as the client -- again, no great loss. Valgrind operates on its own stack, `VG_(stack)`, which it switches to at startup, temporarily switching back to the client's stack when doing system calls for the client.

Valgrind also receives signals on its own stack, `VG_(sigstack)`, but for different gruesome reasons discussed below.

This nice clean switch-back-to-the-real-CPU-whenever-you-like story is muddied by signals. Problem is that signals arrive at arbitrary times and tend to slightly perturb the basic block count, with the result that you can get close to the basic block causing a problem but can't home in on it exactly. My kludgey hack is to define `SIGNAL_SIMULATION` to 1 towards the bottom of `vg_syscall_mem.c`, so that signal handlers are run on the real CPU and don't change the BB counts.

A second hole in the switch-back-to-real-CPU story is that Valgrind's way of delivering signals to the client is different from that of the kernel. Specifically, the layout of the signal delivery frame, and the mechanism used to detect a sighandler returning, are different. So you can't expect to make the transition inside a sighandler and still have things working, but in practice that's not much of a restriction.

Valgrind's implementation of `malloc`, `free`, etc, (in `vg_clientmalloc.c`, not the low-level stuff in `vg_malloc2.c`) is somewhat complicated by the need to handle switching back at arbitrary points. It does work tho.

1.1.4. Correctness

There's only one of me, and I have a Real Life (tm) as well as hacking Valgrind [allegedly :-]. That means I don't have time to waste chasing endless bugs in Valgrind. My emphasis is therefore on doing everything as simply as possible, with correctness, stability and robustness being the number one priority, more important than performance or functionality. As a result:

- The code is absolutely loaded with assertions, and these are **permanently enabled.** I have no plan to remove or disable them later. Over the past couple of months, as valgrind has become more widely used, they have shown their worth, pulling up various bugs which would otherwise have appeared as hard-to-find segmentation faults.

  I am of the view that it's acceptable to spend 5% of the total running time of your valgrindified program doing assertion checks and other internal sanity checks.

- Aside from the assertions, valgrind contains various sets of internal sanity checks, which get run at varying frequencies during normal operation. `VG_(do_sanity_checks)` runs every 1000 basic blocks, which means 500 to 2000 times/second for typical machines at present. It checks that Valgrind hasn't overrun its private stack, and does some simple checks on the memory permissions maps. Once every 25 calls it does some more extensive checks on those maps. Etc, etc.

  The following components also have sanity check code, which can be enabled to aid debugging:

  - The low-level memory-manager (`VG_(mallocSanityCheckArena)`). This does a complete check of all blocks and chains in an arena, which is very slow. Is not engaged by default.

  - The symbol table reader(s): various checks to ensure uniqueness of mappings; see `VG_(read_symbols)` for a start. Is permanently engaged.
  - The A and V bit tracking stuff in `vg_memory.c`. This can be compiled with cpp symbol `VG_DEBUG_MEMORY` defined, which removes all the fast, optimised cases, and uses simple-but-slow fallbacks instead. Not engaged by default.
  - Ditto `VG_DEBUG_LEAKCHECK`.
  - The JITter parses x86 basic blocks into sequences of UCode instructions. It then sanity checks each one with `VG_(saneUInstr)` and sanity checks the sequence as a whole with `VG_(saneUCodeBlock)`. This stuff is engaged by default, and has caught some way-obscure bugs in the simulated CPU machinery in its time.
  - The system call wrapper does `VG_(first_and_last_secondaries_look_plausible)` after every syscall; this is known to pick up bugs in the syscall wrappers. Engaged by default.
  - The main dispatch loop, in `VG_(dispatch)`, checks that translations do not set `%ebp` to any value different from `VG_EBP_DISPATCH_CHECKED` or `& VG_(baseBlock)`. In effect this test is free, and is permanently engaged.
  - There are a couple of ifdefed-out consistency checks I inserted whilst debugging the new register allocater, `vg_do_register_allocation`.

- I try to avoid techniques, algorithms, mechanisms, etc, for which I can supply neither a convincing argument that they are correct, nor sanity-check code which might pick up bugs in my implementation. I don't always succeed in this, but I try. Basically the idea is: avoid techniques which are, in practice, unverifiable, in some sense. When doing anything, always have in mind: "how can I verify that this is correct?"

Some more specific things are:

- Valgrind runs in the same namespace as the client, at least from `ld.so`'s point of view, and it therefore absolutely had better not export any symbol with a name which could clash with that of the client or any of its libraries. Therefore, all globally visible symbols exported from `valgrind.so` are defined using the `VG_` CPP macro. As you'll see from `vg_constants.h`, this appends some arbitrary prefix to the symbol, in order that it be, we hope, globally unique. Currently the prefix is `vgPlain_`. For convenience there are also `VGM_`, `VGP_` and `VGOFF_`. All locally defined symbols are declared `static` and do not appear in the final shared object.

  To check this, I periodically do `nm valgrind.so | grep " T "`, which shows you all the globally exported text symbols. They should all have an approved prefix, except for those like `malloc`, `free`, etc, which we deliberately want to shadow and take precedence over the same names exported from `glibc.so`, so that valgrind can intercept those calls easily. Similarly, `nm valgrind.so | grep " D "` allows you to find any rogue data-segment symbol names.

- Valgrind tries, and almost succeeds, in being completely independent of all other shared objects, in particular of `glibc.so`. For example, we have our own low-level memory manager in `vg_malloc2.c`, which is a fairly standard malloc/free scheme augmented with arenas, and `vg_mylibc.c` exports reimplementations of various bits and pieces you'd normally get from the C library.

  Why all the hassle? Because imagine the potential chaos of both the simulated and real CPUs executing in `glibc.so`. It just seems simpler and cleaner to be completely self-contained, so that only the simulated CPU visits `glibc.so`. In practice it's not much hassle anyway. Also, valgrind starts up before glibc has a chance to initialise itself, and who knows what difficulties that could lead to. Finally, glibc has definitions for some types, specifically `sigset_t`, which conflict (are different from) the Linux kernel's idea of same. When Valgrind wants to fiddle around with signal stuff, it wants to use the kernel's definitions, not glibc's definitions. So it's simplest just to keep glibc out of the picture entirely.

  To find out which glibc symbols are used by Valgrind, reinstate the link flags `-nostdlib -Wl,-no-undefined`. This causes linking to fail, but will tell you what you depend on. I have mostly, but not entirely, got rid of the glibc dependencies; what remains is, IMO, fairly harmless. AFAIK the current dependencies are: `memset`, `memcmp`, `stat`, `system`, `sbrk`, `setjmp` and `longjmp`.

- Similarly, valgrind should not really import any headers other than the Linux kernel headers, since it knows of no API other than the kernel interface to talk to. At the moment this is really not in a good state, and `vg_syscall_mem` imports, via `vg_unsafe.h`, a significant number of C-library headers so as to know the sizes of various structs passed across the kernel boundary. This is of course completely bogus, since there is no guarantee that the C library's definitions of these structs matches those of the kernel. I have started to sort this out using `vg_kerneliface.h`, into which I had intended to copy all kernel definitions which valgrind could need, but this has not gotten very far. At the moment it mostly contains definitions for `sigset_t` and `struct sigaction`, since the kernel's definition for these really does clash with glibc's. I plan to use a `vki_` prefix on all these types and constants, to denote the fact that they pertain to **V**algrind's **K**ernel **I**nterface.

  Another advantage of having a `vg_kerneliface.h` file is that it makes it simpler to interface to a different kernel. Once can, for example, easily imagine writing a new `vg_kerneliface.h` for FreeBSD, or x86 NetBSD.

1.1.5. Current limitations

Support for weird (non-POSIX) signal stuff is patchy. Does anybody care?

1.2. The instrumenting JITter

This really is the heart of the matter. We begin with various side issues.

1.2.1. Run-time storage, and the use of host registers

Valgrind translates client (original) basic blocks into instrumented basic blocks, which live in the translation cache TC, until either the client finishes or the translations are ejected from TC to make room for newer ones.

Since it generates x86 code in memory, Valgrind has complete control of the use of registers in the translations. Now pay attention. I shall say this only once, and it is important you understand this. In what follows I will refer to registers in the host (real) cpu using their standard names, `%eax`, `%edi`, etc. I refer to registers in the simulated CPU by capitalising them: `%EAX`, `%EDI`, etc. These two sets of registers usually bear no direct relationship to each other; there is no fixed mapping between them. This naming scheme is used fairly consistently in the comments in the sources.

Host registers, once things are up and running, are used as follows:

- `%esp`, the real stack pointer, points somewhere in Valgrind's private stack area, `VG_(stack)` or, transiently, into its signal delivery stack, `VG_(sigstack)`.
- `%edi` is used as a temporary in code generation; it is almost always dead, except when used for the `Left` value-tag operations.
- `%eax`, `%ebx`, `%ecx`, `%edx` and `%esi` are available to Valgrind's register allocator. They are dead (carry unimportant values) in between translations, and are live only in translations. The one exception to this is `%eax`, which, as mentioned far above, has a special significance to the dispatch loop `VG_(dispatch)`: when a translation returns to the dispatch loop, `%eax` is expected to contain the original-code-address of the next translation to run. The register allocator is so good at minimising spill code that using five regs and not having to save/restore `%edi` actually gives better code than allocating to `%edi` as well, but then having to push/pop it around special uses.
- `%ebp` points permanently at `VG_(baseBlock)`. Valgrind's translations are position-independent, partly because this is convenient, but also because translations get moved around in TC as part of the LRUing activity. **All** static entities which need to be referred to from generated code, whether data or helper functions, are stored starting at `VG_(baseBlock)` and are therefore reached by indexing from `%ebp`. There is but one exception, which is that by placing the value `VG_EBP_DISPATCH_CHECKED` in `%ebp` just before a return to the dispatcher, the dispatcher is informed that the next address to run, in `%eax`, requires special treatment.
- The real machine's FPU state is pretty much unimportant, for reasons which will become obvious. Ditto its `%eflags` register.

The state of the simulated CPU is stored in memory, in `VG_(baseBlock)`, which is a block of 200 words IIRC. Recall that `%ebp` points permanently at the start of this block. Function `vg_init_baseBlock` decides what the offsets of various entities in `VG_(baseBlock)` are to be, and allocates word offsets for them. The code generator then emits `%ebp` relative addresses to get at those things. The sequence in which entities are allocated has been carefully chosen so that the 32 most popular entities come first, because this means 8-bit offsets can be used in the generated code.

If I was clever, I could make `%ebp` point 32 words along `VG_(baseBlock)`, so that I'd have another 32 words of short-form offsets available, but that's just complicated, and it's not important -- the first 32 words take 99% (or whatever) of the traffic.

Currently, the sequence of stuff in `VG_(baseBlock)` is as follows:

- 9 words, holding the simulated integer registers, `%EAX` .. `%EDI`, and the simulated flags, `%EFLAGS`.
- Another 9 words, holding the V bit "shadows" for the above 9 regs.
- The **addresses** of various helper routines called from generated code: `VG_(helper_value_check4_fail)`, `VG_(helper_value_check0_fail)`, which register V-check failures, `VG_(helperc_STOREV4)`, `VG_(helperc_STOREV1)`, `VG_(helperc_LOADV4)`, `VG_(helperc_LOADV1)`, which do stores and loads of V bits to/from the sparse array which keeps track of V bits in memory, and `VGM_(handle_esp_assignment)`, which messes with memory addressibility resulting from changes in `%ESP`.
- The simulated `%EIP`.
- 24 spill words, for when the register allocator can't make it work with 5 measly registers.
- Addresses of helpers `VG_(helperc_STOREV2)`, `VG_(helperc_LOADV2)`. These are here because 2-byte loads and stores are relatively rare, so are placed above the magic 32-word offset boundary.

- For similar reasons, addresses of helper functions `VGM_(fpu_write_check)` and `VGM_(fpu_read_check)`, which handle the A/V maps testing and changes required by FPU writes/reads.
- Some other boring helper addresses: `VG_(helper_value_check2_fail)` and `VG_(helper_value_check1_fail)`. These are probably never emitted now, and should be removed.
- The entire state of the simulated FPU, which I believe to be 108 bytes long.
- Finally, the addresses of various other helper functions in `vg_helpers.S`, which deal with rare situations which are tedious or difficult to generate code in-line for.

As a general rule, the simulated machine's state lives permanently in memory at `VG_(baseBlock)`. However, the JITter does some optimisations which allow the simulated integer registers to be cached in real registers over multiple simulated instructions within the same basic block. These are always flushed back into memory at the end of every basic block, so that the in-memory state is up-to-date between basic blocks. (This flushing is implied by the statement above that the real machine's allocatable registers are dead in between simulated blocks).

1.2.2. Startup, shutdown, and system calls

Getting into of Valgrind (`VG_(startup)`, called from `valgrind.so`'s initialisation section), really means copying the real CPU's state into `VG_(baseBlock)`, and then installing our own stack pointer, etc, into the real CPU, and then starting up the JITter. Exiting valgrind involves copying the simulated state back to the real state.

Unfortunately, there's a complication at startup time. Problem is that at the point where we need to take a snapshot of the real CPU's state, the offsets in `VG_(baseBlock)` are not set up yet, because to do so would involve disrupting the real machine's state significantly. The way round this is to dump the real machine's state into a temporary, static block of memory, `VG_(m_state_static)`. We can then set up the `VG_(baseBlock)` offsets at our leisure, and copy into it from `VG_(m_state_static)` at some convenient later time. This copying is done by `VG_(copy_m_state_static_to_baseBlock)`.

On exit, the inverse transformation is (rather unnecessarily) used: stuff in `VG_(baseBlock)` is copied to `VG_(m_state_static)`, and the assembly stub then copies from `VG_(m_state_static)` into the real machine registers.

Doing system calls on behalf of the client (`vg_syscall.S`) is something of a half-way house. We have to make the world look sufficiently like that which the client would normally have to make the syscall actually work properly, but we can't afford to lose control. So the trick is to copy all of the client's state, **except its program counter**, into the real CPU, do the system call, and copy the state back out. Note that the client's state includes its stack pointer register, so one effect of this partial restoration is to cause the system call to be run on the client's stack, as it should be.

As ever there are complications. We have to save some of our own state somewhere when restoring the client's state into the CPU, so that we can keep going sensibly afterwards. In fact the only thing which is important is our own stack pointer, but for paranoia reasons I save and restore our own FPU state as well, even though that's probably pointless.

The complication on the above complication is, that for horrible reasons to do with signals, we may have to handle a second client system call whilst the client is blocked inside some other system call (unbelievable!). That means there's two sets of places to dump Valgrind's stack pointer and FPU state across the syscall, and we decide which to use by consulting `VG_(syscall_depth)`, which is in turn maintained by `VG_(wrap_syscall)`.

1.2.3. Introduction to UCode

UCode lies at the heart of the x86-to-x86 JITter. The basic premise is that dealing the the x86 instruction set head-on is just too darn complicated, so we do the traditional compiler-writer's trick and translate it into a simpler, easier-to-deal-with form.

In normal operation, translation proceeds through six stages, coordinated by `VG_(translate)`:

1. Parsing of an x86 basic block into a sequence of UCode instructions (`VG_(disBB)`).
2. UCode optimisation (`vg_improve`), with the aim of caching simulated registers in real registers over multiple simulated instructions, and removing redundant simulated `%EFLAGS` saving/restoring.
3. UCode instrumentation (`vg_instrument`), which adds value and address checking code.
4. Post-instrumentation cleanup (`vg_cleanup`), removing redundant value-check computations.
5. Register allocation (`vg_do_register_allocation`), which, note, is done on UCode.
6. Emission of final instrumented x86 code (`VG_(emit_code)`).

Notice how steps 2, 3, 4 and 5 are simple UCode-to-UCode transformation passes, all on straight-line blocks of UCode (type `UCodeBlock`). Steps 2 and 4 are optimisation passes and can be disabled for debugging purposes, with `--optimise=no` and `--cleanup=no` respectively.

Valgrind can also run in a no-instrumentation mode, given `--instrument=no`. This is useful for debugging the JITter quickly without having to deal with the complexity of the instrumentation mechanism too. In this mode, steps 3 and 4 are omitted.

These flags combine, so that `--instrument=no` together with `--optimise=no` means only steps 1, 5 and 6 are used. `--single-step=yes` causes each x86 instruction to be treated as a single basic block. The translations are terrible but this is sometimes instructive.

The `--stop-after=N` flag switches back to the real CPU after `N` basic blocks. It also re-JITs the final basic block executed and prints the debugging info resulting, so this gives you a way to get a quick snapshot of how a basic block looks as it passes through the six stages mentioned above. If you want to see full information for every block translated (probably not, but still ...) find, in `VG_(translate)`, the lines

```
  dis = True;
  dis = debugging_translation;
```

and comment out the second line. This will spew out debugging junk faster than you can possibly imagine.

1.2.4. UCode operand tags: type `Tag`

UCode is, more or less, a simple two-address RISC-like code. In keeping with the x86 AT&T assembly syntax, generally speaking the first operand is the source operand, and the second is the destination operand, which is modified when the uinstr is notionally executed.

UCode instructions have up to three operand fields, each of which has a corresponding `Tag` describing it. Possible values for the tag are:

- `NoValue`: indicates that the field is not in use.
- `Lit16`: the field contains a 16-bit literal.
- `Literal`: the field denotes a 32-bit literal, whose value is stored in the `lit32` field of the uinstr itself. Since there is only one `lit32` for the whole uinstr, only one operand field may contain this tag.

- `SpillNo`: the field contains a spill slot number, in the range 0 to 23 inclusive, denoting one of the spill slots contained inside `VG_(baseBlock)`. Such tags only exist after register allocation.
- `RealReg`: the field contains a number in the range 0 to 7 denoting an integer x86 ("real") register on the host. The number is the Intel encoding for integer registers. Such tags only exist after register allocation.
- `ArchReg`: the field contains a number in the range 0 to 7 denoting an integer x86 register on the simulated CPU. In reality this means a reference to one of the first 8 words of `VG_(baseBlock)`. Such tags can exist at any point in the translation process.
- Last, but not least, `TempReg`. The field contains the number of one of an infinite set of virtual (integer) registers. `TempReg`s are used everywhere throughout the translation process; you can have as many as you want. The register allocator maps as many as it can into `RealReg`s and turns the rest into `SpillNo`s, so `TempReg`s should not exist after the register allocation phase.

  `TempReg`s are always 32 bits long, even if the data they hold is logically shorter. In that case the upper unused bits are required, and, I think, generally assumed, to be zero. `TempReg`s holding V bits for quantities shorter than 32 bits are expected to have ones in the unused places, since a one denotes "undefined".

1.2.5. UCode instructions: type `UInstr`

UCode was carefully designed to make it possible to do register allocation on UCode and then translate the result into x86 code without needing any extra registers ... well, that was the original plan, anyway. Things have gotten a little more complicated since then. In what follows, UCode instructions are referred to as uinstrs, to distinguish them from x86 instructions. Uinstrs of course have uopcodes which are (naturally) different from x86 opcodes.

A uinstr (type `UInstr`) contains various fields, not all of which are used by any one uopcode:

- Three 16-bit operand fields, `val1`, `val2` and `val3`.
- Three tag fields, `tag1`, `tag2` and `tag3`. Each of these has a value of type `Tag`, and they describe what the `val1`, `val2` and `val3` fields contain.
- A 32-bit literal field.
- Two `FlagSet`s, specifying which x86 condition codes are read and written by the uinstr.
- An opcode byte, containing a value of type `Opcode`.
- A size field, indicating the data transfer size (1/2/4/8/10) in cases where this makes sense, or zero otherwise.
- A condition-code field, which, for jumps, holds a value of type `Condcode`, indicating the condition which applies. The encoding is as it is in the x86 insn stream, except we add a 17th value `CondAlways` to indicate an unconditional transfer.
- Various 1-bit flags, indicating whether this insn pertains to an x86 CALL or RET instruction, whether a widening is signed or not, etc.

UOpcodes (type `Opcode`) are divided into two groups: those necessary merely to express the functionality of the x86 code, and extra uopcodes needed to express the instrumentation. The former group contains:

- `GET` and `PUT`, which move values from the simulated CPU's integer registers (`ArchReg`s) into `TempReg`s, and back. `GETF` and `PUTF` do the corresponding thing for the simulated `%EFLAGS`. There are no corresponding insns for the FPU register stack, since we don't explicitly simulate its registers.

- `LOAD` and `STORE`, which, in RISC-like fashion, are the only uinstrs able to interact with memory.

- `MOV` and `CMOV` allow unconditional and conditional moves of values between `TempReg`s.

- ALU operations. Again in RISC-like fashion, these only operate on `TempReg`s (before reg-alloc) or `RealReg`s (after reg-alloc). These are: `ADD`, `ADC`, `AND`, `OR`, `XOR`, `SUB`, `SBB`, `SHL`, `SHR`, `SAR`, `ROL`, `ROR`, `RCL`, `RCR`, `NOT`, `NEG`, `INC`, `DEC`, `BSWAP`, `CC2VAL` and `WIDEN`. `WIDEN` does signed or unsigned value widening. `CC2VAL` is used to convert condition codes into a value, zero or one. The rest are obvious.

  To allow for more efficient code generation, we bend slightly the restriction at the start of the previous para: for `ADD`, `ADC`, `XOR`, `SUB` and `SBB`, we allow the first (source) operand to also be an `ArchReg`, that is, one of the simulated machine's registers. Also, many of these ALU ops allow the source operand to be a literal. See `VG_(saneUInstr)` for the final word on the allowable forms of uinstrs.

- `LEA1` and `LEA2` are not strictly necessary, but allow faciliate better translations. They record the fancy x86 addressing modes in a direct way, which allows those amodes to be emitted back into the final instruction stream more or less verbatim.

- `CALLM` calls a machine-code helper, one of the methods whose address is stored at some `VG_(baseBlock)` offset. `PUSH` and `POP` move values to/from `TempReg` to the real (Valgrind's) stack, and `CLEAR` removes values from the stack. `CALLM_S` and `CALLM_E` delimit the boundaries of call setups and clearings, for the benefit of the instrumentation passes. Getting this right is critical, and so `VG_(saneUCodeBlock)` makes various checks on the use of these uopcodes.

  It is important to understand that these uopcodes have nothing to do with the x86 `call`, `return, push` or `pop` instructions, and are not used to implement them. Those guys turn into combinations of `GET`, `PUT`, `LOAD`, `STORE`, `ADD`, `SUB`, and `JMP`. What these uopcodes support is calling of helper functions such as `VG_(helper_imul_32_64)`, which do stuff which is too difficult or tedious to emit inline.

- `FPU`, `FPU_R` and `FPU_W`. Valgrind doesn't attempt to simulate the internal state of the FPU at all. Consequently it only needs to be able to distinguish FPU ops which read and write memory from those that don't, and for those which do, it needs to know the effective address and data transfer size. This is made easier because the x86 FP instruction encoding is very regular, basically consisting of 16 bits for a non-memory FPU insn and 11 (IIRC) bits + an address mode for a memory FPU insn. So our `FPU` uinstr carries the 16 bits in its `val1` field. And `FPU_R` and `FPU_W` carry 11 bits in that field, together with the identity of a `TempReg` or (later) `RealReg` which contains the address.

- `JIFZ` is unique, in that it allows a control-flow transfer which is not deemed to end a basic block. It causes a jump to a literal (original) address if the specified argument is zero.

- Finally, `INCEIP` advances the simulated `%EIP` by the specified literal amount. This supports lazy `%EIP` updating, as described below.

Stages 1 and 2 of the 6-stage translation process mentioned above deal purely with these uopcodes, and no others. They are sufficient to express pretty much all the x86 32-bit protected-mode instruction set, at least everything understood by a pre-MMX original Pentium (P54C).

Stages 3, 4, 5 and 6 also deal with the following extra "instrumentation" uopcodes. They are used to express all the definedness-tracking and -checking machinery which valgrind does. In later sections we show how to create checking code for each of the uopcodes above. Note that these instrumentation uopcodes, although some appearing complicated, have been carefully chosen so that efficient x86 code can be generated for them. GNU superopt v2.5 did a great job helping out here. Anyways, the uopcodes are as follows:

- `GETV` and `PUTV` are analogues to `GET` and `PUT` above. They are identical except that they move the V bits for the specified values back and forth to `TempRegs`, rather than moving the values themselves.
- Similarly, `LOADV` and `STOREV` read and write V bits from the synthesised shadow memory that Valgrind maintains. In fact they do more than that, since they also do address-validity checks, and emit complaints if the read/written addresses are unaddressible.
- `TESTV`, whose parameters are a `TempReg` and a size, tests the V bits in the `TempReg`, at the specified operation size (0/1/2/4 byte) and emits an error if any of them indicate undefinedness. This is the only uopcode capable of doing such tests.
- `SETV`, whose parameters are also `TempReg` and a size, makes the V bits in the `TempReg` indicated definedness, at the specified operation size. This is usually used to generate the correct V bits for a literal value, which is of course fully defined.
- `GETVF` and `PUTVF` are analogues to `GETF` and `PUTF`. They move the single V bit used to model definedness of `%EFLAGS` between its home in `VG_(baseBlock)` and the specified `TempReg`.
- `TAG1` denotes one of a family of unary operations on `TempReg`s containing V bits. Similarly, `TAG2` denotes one in a family of binary operations on V bits.

These 10 uopcodes are sufficient to express Valgrind's entire definedness-checking semantics. In fact most of the interesting magic is done by the `TAG1` and `TAG2` suboperations.

First, however, I need to explain about V-vector operation sizes. There are 4 sizes: 1, 2 and 4, which operate on groups of 8, 16 and 32 V bits at a time, supporting the usual 1, 2 and 4 byte x86 operations. However there is also the mysterious size 0, which really means a single V bit. Single V bits are used in various circumstances; in particular, the definedness of `%EFLAGS` is modelled with a single V bit. Now might be a good time to also point out that for V bits, 1 means "undefined" and 0 means "defined". Similarly, for A bits, 1 means "invalid address" and 0 means "valid address". This seems counterintuitive (and so it is), but testing against zero on x86s saves instructions compared to testing against all 1s, because many ALU operations set the Z flag for free, so to speak.

With that in mind, the tag ops are:

- **(UNARY) Pessimising casts:** `VgT_PCast40`, `VgT_PCast20`, `VgT_PCast10`, `VgT_PCast01`, `VgT_PCast02` and `VgT_PCast04`. A "pessimising cast" takes a V-bit vector at one size, and creates a new one at another size, pessimised in the sense that if any of the bits in the source vector indicate undefinedness, then all the bits in the result indicate undefinedness. In this case the casts are all to or from a single V bit, so for example `VgT_PCast40` is a pessimising cast from 32 bits to 1, whereas `VgT_PCast04` simply copies the single source V bit into all 32 bit positions in the result. Surprisingly, these ops can all be implemented very efficiently.

  There are also the pessimising casts `VgT_PCast14`, from 8 bits to 32, `VgT_PCast12`, from 8 bits to 16, and `VgT_PCast11`, from 8 bits to 8. This last one seems nonsensical, but in fact it isn't a no-op because, as mentioned above, any undefined (1) bits in the source infect the entire result.
- **(UNARY) Propagating undefinedness upwards in a word:** `VgT_Left4`, `VgT_Left2` and `VgT_Left1`. These are used to simulate the worst-case effects of carry propagation in adds and subtracts. They return a V vector identical to the original, except that if the original contained any undefined bits, then it and all bits above it are marked as undefined too. Hence the Left bit in the names.

- **(UNARY) Signed and unsigned value widening:** `VgT_SWiden14`, `VgT_SWiden24`, `VgT_SWiden12`, `VgT_ZWiden14`, `VgT_ZWiden24` and `VgT_ZWiden12`. These mimic the definedness effects of standard signed and unsigned integer widening. Unsigned widening creates zero bits in the new positions, so `VgT_ZWiden*` accordingly park mark those parts of their argument as defined. Signed widening copies the sign bit into the new positions, so `VgT_SWiden*` copies the definedness of the sign bit into the new positions. Because 1 means undefined and 0 means defined, these operations can (fascinatingly) be done by the same operations which they mimic. Go figure.

- **(BINARY) Undefined-if-either-Undefined, Defined-if-either-Defined:** `VgT_UifU4`, `VgT_UifU2`, `VgT_UifU1`, `VgT_UifU0`, `VgT_DifD4`, `VgT_DifD2`, `VgT_DifD1`. These do simple bitwise operations on pairs of V-bit vectors, with `UifU` giving undefined if either arg bit is undefined, and `DifD` giving defined if either arg bit is defined. Abstract interpretation junkies, if any make it this far, may like to think of them as meets and joins (or is it joins and meets) in the definedness lattices.

- **(BINARY; one value, one V bits) Generate argument improvement terms for AND and OR.** `VgT_ImproveAND4_TQ`, `VgT_ImproveAND2_TQ`, `VgT_ImproveAND1_TQ`, `VgT_ImproveOR4_TQ`, `VgT_ImproveOR2_TQ`, `VgT_ImproveOR1_TQ`. These help out with AND and OR operations. AND and OR have the inconvenient property that the definedness of the result depends on the actual values of the arguments as well as their definedness. At the bit level:

  ```
  1 AND undefined = undefined, but
  0 AND undefined = 0, and
  similarly
  0 OR undefined = undefined, but
  1 OR undefined = 1.
  ```

  It turns out that gcc (quite legitimately) generates code which relies on this fact, so we have to model it properly in order to avoid flooding users with spurious value errors. The ultimate definedness result of AND and OR is calculated using `UifU` on the definedness of the arguments, but we also `DifD` in some "improvement" terms which take into account the above phenomena.

  `ImproveAND` takes as its first argument the actual value of an argument to AND (the T) and the definedness of that argument (the Q), and returns a V-bit vector which is defined (0) for bits which have value 0 and are defined; this, when `DifD` into the final result causes those bits to be defined even if the corresponding bit in the other argument is undefined.

  The `ImproveOR` ops do the dual thing for OR arguments. Note that XOR does not have this property that one argument can make the other irrelevant, so there is no need for such complexity for XOR.

That’s all the tag ops. If you stare at this long enough, and then run Valgrind and stare at the pre- and post-instrumented ucode, it should be fairly obvious how the instrumentation machinery hangs together.

One point, if you do this: in order to make it easy to differentiate `TempReg`s carrying values from `TempReg`s carrying V bit vectors, Valgrind prints the former as (for example) `t28` and the latter as `q28`; the fact that they carry the same number serves to indicate their relationship. This is purely for the convenience of the human reader; the register allocator and code generator don’t regard them as different.

1.2.6. Translation into UCode

`VG_(disBB)` allocates a new `UCodeBlock` and then uses `disInstr` to translate x86 instructions one at a time into UCode, dumping the result in the `UCodeBlock`. This goes on until a control-flow transfer instruction is encountered.

Despite the large size of `vg_to_ucode.c`, this translation is really very simple. Each x86 instruction is translated entirely independently of its neighbours, merrily allocating new `TempReg`s as it goes. The idea is to have a simple translator -- in reality, no more than a macro-expander -- and the -- resulting bad UCode translation is cleaned up by the UCode optimisation phase which follows. To give you an idea of some x86 instructions and their translations (this is a complete basic block, as Valgrind sees it):

```
0x40435A50:  incl %edx
    0: GETL     %EDX, t0
    1: INCL     t0  (-wOSZAP)
    2: PUTL     t0, %EDX

0x40435A51:  movsbl (%edx),%eax
    3: GETL     %EDX, t2
    4: LDB     (t2), t2
    5: WIDENL_Bs t2
    6: PUTL     t2, %EAX

0x40435A54:  testb $0x20, 1(%ecx,%eax,2)
    7: GETL     %EAX, t6
    8: GETL     %ECX, t8
    9: LEA2L    1(t8,t6,2), t4
   10: LDB      (t4), t10
   11: MOVB     $0x20, t12
   12: ANDB     t12, t10  (-wOSZACP)
   13: INCEIPo   $9

0x40435A59:  jnz-8 0x40435A50
   14: Jnzo     $0x40435A50  (-rOSZACP)
   15: JMPo     $0x40435A5B
```

Notice how the block always ends with an unconditional jump to the next block. This is a bit unnecessary, but makes many things simpler.

Most x86 instructions turn into sequences of `GET`, `PUT`, `LEA1`, `LEA2`, `LOAD` and `STORE`. Some complicated ones however rely on calling helper bits of code in `vg_helpers.S`. The ucode instructions `PUSH`, `POP`, `CALL`, `CALLM_S` and `CALLM_E` support this. The calling convention is somewhat ad-hoc and is not the C calling convention. The helper routines must save all integer registers, and the flags, that they use. Args are passed on the stack underneath the return address, as usual, and if result(s) are to be returned, it (they) are either placed in dummy arg slots created by the ucode `PUSH` sequence, or just overwrite the incoming args.

In order that the instrumentation mechanism can handle calls to these helpers, `VG_(saneUCodeBlock)` enforces the following restrictions on calls to helpers:

- Each `CALL` uinstr must be bracketed by a preceding `CALLM_S` marker (dummy uinstr) and a trailing `CALLM_E` marker. These markers are used by the instrumentation mechanism later to establish the boundaries of the `PUSH`, `POP` and `CLEAR` sequences for the call.
- `PUSH`, `POP` and `CLEAR` may only appear inside sections bracketed by `CALLM_S` and `CALLM_E`, and nowhere else.
- In any such bracketed section, no two `PUSH` insns may push the same `TempReg`. Dually, no two two `POP`s may pop the same `TempReg`.

- Finally, although this is not checked, args should be removed from the stack with `CLEAR`, rather than `POP`s into a `TempReg` which is not subsequently used. This is because the instrumentation mechanism assumes that all values `POP`ped from the stack are actually used.

Some of the translations may appear to have redundant `TempReg`-to-`TempReg` moves. This helps the next phase, UCode optimisation, to generate better code.

1.2.7. UCode optimisation

UCode is then subjected to an improvement pass (`vg_improve()`), which blurs the boundaries between the translations of the original x86 instructions. It's pretty straightforward. Three transformations are done:

- Redundant `GET` elimination. Actually, more general than that -- eliminates redundant fetches of ArchRegs. In our running example, uinstr 3 `GET`s `%EDX` into `t2` despite the fact that, by looking at the previous uinstr, it is already in `t0`. The `GET` is therefore removed, and `t2` renamed to `t0`. Assuming `t0` is allocated to a host register, it means the simulated `%EDX` will exist in a host CPU register for more than one simulated x86 instruction, which seems to me to be a highly desirable property.

  There is some mucking around to do with subregisters; `%AL` vs `%AH` `%AX` vs `%EAX` etc. I can't remember how it works, but in general we are very conservative, and these tend to invalidate the caching.

- Redundant `PUT` elimination. This annuls `PUT`s of values back to simulated CPU registers if a later `PUT` would overwrite the earlier `PUT` value, and there is no intervening reads of the simulated register (`ArchReg`).

  As before, we are paranoid when faced with subregister references. Also, `PUT`s of `%ESP` are never annulled, because it is vital the instrumenter always has an up-to-date `%ESP` value available, `%ESP` changes affect addressibility of the memory around the simulated stack pointer.

  The implication of the above paragraph is that the simulated machine's registers are only lazily updated once the above two optimisation phases have run, with the exception of `%ESP`. `TempReg`s go dead at the end of every basic block, from which is is inferrable that any `TempReg` caching a simulated CPU reg is flushed (back into the relevant `VG_(baseBlock)` slot) at the end of every basic block. The further implication is that the simulated registers are only up-to-date at in between basic blocks, and not at arbitrary points inside basic blocks. And the consequence of that is that we can only deliver signals to the client in between basic blocks. None of this seems any problem in practice.

- Finally there is a simple def-use thing for condition codes. If an earlier uinstr writes the condition codes, and the next uinsn along which actually cares about the condition codes writes the same or larger set of them, but does not read any, the earlier uinsn is marked as not writing any condition codes. This saves a lot of redundant cond-code saving and restoring.

The effect of these transformations on our short block is rather unexciting, and shown below. On longer basic blocks they can dramatically improve code quality.

```
at 3: delete GET, rename t2 to t0 in (4 .. 6)
at 7: delete GET, rename t6 to t0 in (8 .. 9)
at 1: annul flag write OSZAP due to later OSZACP

Improved code:
    0: GETL     %EDX, t0
    1: INCL     t0
    2: PUTL     t0, %EDX
    4: LDB      (t0), t0
    5: WIDENL_Bs t0
    6: PUTL     t0, %EAX
    8: GETL     %ECX, t8
    9: LEA2L    1(t8,t0,2), t4
   10: LDB      (t4), t10
   11: MOVB     $0x20, t12
   12: ANDB     t12, t10  (-wOSZACP)
   13: INCEIPo  $9
   14: Jnzo     $0x40435A50  (-rOSZACP)
   15: JMPo     $0x40435A5B
```

1.2.8. UCode instrumentation

Once you understand the meaning of the instrumentation uinstrs, discussed in detail above, the instrumentation scheme is fairly straightforward. Each uinstr is instrumented in isolation, and the instrumentation uinstrs are placed before the original uinstr. Our running example continues below. I have placed a blank line after every original ucode, to make it easier to see which instrumentation uinstrs correspond to which originals.

As mentioned somewhere above, `TempReg`s carrying values have names like `t28`, and each one has a shadow carrying its V bits, with names like `q28`. This pairing aids in reading instrumented ucode.

One decision about all this is where to have "observation points", that is, where to check that V bits are valid. I use a minimalistic scheme, only checking where a failure of validity could cause the original program to (seg)fault. So the use of values as memory addresses causes a check, as do conditional jumps (these cause a check on the definedness of the condition codes). And arguments `PUSH`ed for helper calls are checked, hence the weird restrictions on help call preambles described above.

Another decision is that once a value is tested, it is thereafter regarded as defined, so that we do not emit multiple undefined-value errors for the same undefined value. That means that `TESTV` uinstrs are always followed by `SETV` on the same (shadow) `TempReg`s. Most of these `SETV`s are redundant and are removed by the post-instrumentation cleanup phase.

The instrumentation for calling helper functions deserves further comment. The definedness of results from a helper is modelled using just one V bit. So, in short, we do pessimising casts of the definedness of all the args, down to a single bit, and then `UifU` these bits together. So this single V bit will say "undefined" if any part of any arg is undefined. This V bit is then pessimally cast back up to the result(s) sizes, as needed. If, by seeing that all the args are got rid of with `CLEAR` and none with `POP`, Valgrind sees that the result of the call is not actually used, it immediately examines the result V bit with a `TESTV` -- `SETV` pair. If it did not do this, there would be no observation point to detect that the some of the args to the helper were undefined. Of course, if the helper's results are indeed used, we don't do this, since the result usage will presumably cause the result definedness to be checked at some suitable future point.

In general Valgrind tries to track definedness on a bit-for-bit basis, but as the above para shows, for calls to helpers we throw in the towel and approximate down to a single bit. This is because it's too complex and difficult to track bit-level definedness through complex ops such as integer multiply and divide, and in any case there is no reasonable code fragments which attempt to (eg) multiply two partially-defined values and end up with something meaningful, so there seems little point in modelling multiplies, divides, etc, in that level of detail.

Integer loads and stores are instrumented with firstly a test of the definedness of the address, followed by a `LOADV` or `STOREV` respectively. These turn into calls to (for example) `VG_(helperc_LOADV4)`. These helpers do two things: they perform an address-valid check, and they load or store V bits from/to the relevant address in the (simulated V-bit) memory.

FPU loads and stores are different. As above the definedness of the address is first tested. However, the helper routine for FPU loads (`VGM_(fpu_read_check)`) emits an error if either the address is invalid or the referenced area contains undefined values. It has to do this because we do not simulate the FPU at all, and so cannot track definedness of values loaded into it from memory, so we have to check them as soon as they are loaded into the FPU, ie, at this point. We notionally assume that everything in the FPU is defined.

It follows therefore that FPU writes first check the definedness of the address, then the validity of the address, and finally mark the written bytes as well-defined.

If anyone is inspired to extend Valgrind to MMX/SSE insns, I suggest you use the same trick. It works provided that the FPU/MMX unit is not used to merely as a conduit to copy partially undefined data from one place in memory to another. Unfortunately the integer CPU is used like that (when copying C structs with holes, for example) and this is the cause of much of the elaborateness of the instrumentation here described.

`vg_instrument()` in `vg_translate.c` actually does the instrumentation. There are comments explaining how each uinstr is handled, so we do not repeat that here. As explained already, it is bit-accurate, except for calls to helper functions. Unfortunately the x86 insns `bt/bts/btc/btr` are done by helper fns, so bit-level accuracy is lost there. This should be fixed by doing them inline; it will probably require adding a couple new uinstrs. Also, left and right rotates through the carry flag (x86 `rcl` and `rcr`) are approximated via a single V bit; so far this has not caused anyone to complain. The non-carry rotates, `rol` and `ror`, are much more common and are done exactly. Re-visiting the instrumentation for AND and OR, they seem rather verbose, and I wonder if it could be done more concisely now.

The lowercase `o` on many of the uopcodes in the running example indicates that the size field is zero, usually meaning a single-bit operation.

Anyroads, the post-instrumented version of our running example looks like this:

```
Instrumented code:
    0: GETVL    %EDX, q0
    1: GETL     %EDX, t0

    2: TAG1o    q0 = Left4 ( q0 )
    3: INCL     t0

    4: PUTVL    q0, %EDX
    5: PUTL     t0, %EDX

    6: TESTVL   q0
    7: SETVL    q0
    8: LOADVB   (t0), q0
    9: LDB      (t0), t0

   10: TAG1o    q0 = SWiden14 ( q0 )
   11: WIDENL_Bs t0

   12: PUTVL    q0, %EAX
   13: PUTL     t0, %EAX

   14: GETVL    %ECX, q8
   15: GETL     %ECX, t8

   16: MOVL     q0, q4
   17: SHLL     $0x1, q4
   18: TAG2o    q4 = UifU4 ( q8, q4 )
   19: TAG1o    q4 = Left4 ( q4 )
   20: LEA2L    1(t8,t0,2), t4

   21: TESTVL   q4
   22: SETVL    q4
   23: LOADVB   (t4), q10
   24: LDB      (t4), t10

   25: SETVB    q12
   26: MOVB     $0x20, t12

   27: MOVL     q10, q14
   28: TAG2o    q14 = ImproveAND1_TQ ( t10, q14 )
   29: TAG2o    q10 = UifU1 ( q12, q10 )
   30: TAG2o    q10 = DifD1 ( q14, q10 )
   31: MOVL     q12, q14
   32: TAG2o    q14 = ImproveAND1_TQ ( t12, q14 )
   33: TAG2o    q10 = DifD1 ( q14, q10 )
   34: MOVL     q10, q16
   35: TAG1o    q16 = PCast10 ( q16 )
   36: PUTVFo   q16
   37: ANDB     t12, t10  (-wOSZACP)

   38: INCEIPo  $9

   39: GETVFo   q18
   40: TESTVo   q18
   41: SETVo    q18
   42: Jnzo     $0x40435A50  (-rOSZACP)

   43: JMPo     $0x40435A5B
```

1.2.9. UCode post-instrumentation cleanup

This pass, coordinated by `vg_cleanup()`, removes redundant definedness computation created by the simplistic instrumentation pass. It consists of two passes, `vg_propagate_definedness()` followed by `vg_delete_redundant_SETVs`.

`vg_propagate_definedness()` is a simple constant-propagation and constant-folding pass. It tries to determine which `TempReg`s containing V bits will always indicate "fully defined", and it propagates this information as far as it can, and folds out as many operations as possible. For example, the instrumentation for an ADD of a literal to a variable quantity will be reduced down so that the definedness of the result is simply the definedness of the variable quantity, since the literal is by definition fully defined.

`vg_delete_redundant_SETVs` removes `SETV`s on shadow `TempReg`s for which the next action is a write. I don't think there's anything else worth saying about this; it is simple. Read the sources for details.

So the cleaned-up running example looks like this. As above, I have inserted line breaks after every original (non-instrumentation) uinstr to aid readability. As with straightforward ucode optimisation, the results in this block are undramatic because it is so short; longer blocks benefit more because they have more redundancy which gets eliminated.

```
at 29: delete UifU1 due to defd arg1
at 32: change ImproveAND1_TQ to MOV due to defd arg2
at 41: delete SETV
at 31: delete MOV
at 25: delete SETV
at 22: delete SETV
at 7: delete SETV

     0: GETVL    %EDX, q0
     1: GETL     %EDX, t0

     2: TAG1o    q0 = Left4 ( q0 )
     3: INCL     t0

     4: PUTVL    q0, %EDX
     5: PUTL     t0, %EDX

     6: TESTVL   q0
     8: LOADVB   (t0), q0
     9: LDB      (t0), t0

    10: TAG1o    q0 = SWiden14 ( q0 )
    11: WIDENL_Bs t0

    12: PUTVL    q0, %EAX
    13: PUTL     t0, %EAX

    14: GETVL    %ECX, q8
    15: GETL     %ECX, t8

    16: MOVL     q0, q4
    17: SHLL     $0x1, q4
    18: TAG2o    q4 = UifU4 ( q8, q4 )
    19: TAG1o    q4 = Left4 ( q4 )
    20: LEA2L    1(t8,t0,2), t4

    21: TESTVL   q4
    23: LOADVB   (t4), q10
    24: LDB      (t4), t10

    26: MOVB     $0x20, t12

    27: MOVL     q10, q14
    28: TAG2o    q14 = ImproveAND1_TQ ( t10, q14 )
    30: TAG2o    q10 = DifD1 ( q14, q10 )
    32: MOVL     t12, q14
    33: TAG2o    q10 = DifD1 ( q14, q10 )
    34: MOVL     q10, q16
    35: TAG1o    q16 = PCast10 ( q16 )
    36: PUTVFo   q16
    37: ANDB     t12, t10  (-wOSZACP)

    38: INCEIPo  $9
    39: GETVFo   q18
    40: TESTVo   q18
    42: Jnzo     $0x40435A50  (-rOSZACP)

    43: JMPo     $0x40435A5B
```

1.2.10. Translation from UCode

This is all very simple, even though `vg_from_ucode.c` is a big file. Position-independent x86 code is generated into a dynamically allocated array `emitted_code`; this is doubled in size when it overflows. Eventually the array is handed back to the caller of `VG_(translate)`, who must copy the result into TC and TT, and free the array.

This file is structured into four layers of abstraction, which, thankfully, are glued back together with extensive `__inline__` directives. From the bottom upwards:

- Address-mode emitters, `emit_amode_regmem_reg` et al.
- Emitters for specific x86 instructions. There are quite a lot of these, with names such as `emit_movv_offregmem_reg`. The `v` suffix is Intel parlance for a 16/32 bit insn; there are also `b` suffixes for 8 bit insns.
- The next level up are the `synth_*` functions, which synthesise possibly a sequence of raw x86 instructions to do some simple task. Some of these are quite complex because they have to work around Intel's silly restrictions on subregister naming. See `synth_nonshiftop_reg_reg` for example.
- Finally, at the top of the heap, we have `emitUInstr()`, which emits code for a single uinstr.

Some comments:

- The hack for FPU instructions becomes apparent here. To do a `FPU` ucode instruction, we load the simulated FPU's state into from its `VG_(baseBlock)` into the real FPU using an x86 `frstor` insn, do the ucode `FPU` insn on the real CPU, and write the updated FPU state back into `VG_(baseBlock)` using an `fnsave` instruction. This is pretty brutal, but is simple and it works, and even seems tolerably efficient. There is no attempt to cache the simulated FPU state in the real FPU over multiple back-to-back ucode FPU instructions.

  `FPU_R` and `FPU_W` are also done this way, with the minor complication that we need to patch in some addressing mode bits so the resulting insn knows the effective address to use. This is easy because of the regularity of the x86 FPU instruction encodings.
- An analogous trick is done with ucode insns which claim, in their `flags_r` and `flags_w` fields, that they read or write the simulated `%EFLAGS`. For such cases we first copy the simulated `%EFLAGS` into the real `%eflags`, then do the insn, then, if the insn says it writes the flags, copy back to `%EFLAGS`. This is a bit expensive, which is why the ucode optimisation pass goes to some effort to remove redundant flag-update annotations.

And so ... that's the end of the documentation for the instrumentating translator! It's really not that complex, because it's composed as a sequence of simple(ish) self-contained transformations on straight-line blocks of code.

1.2.11. Top-level dispatch loop

Urk. In `VG_(toploop)`. This is basically boring and unsurprising, not to mention fiddly and fragile. It needs to be cleaned up.

The only perhaps surprise is that the whole thing is run on top of a `setjmp`-installed exception handler, because, supposing a translation got a segfault, we have to bail out of the Valgrind-supplied exception handler `VG_(oursignalhandler)` and immediately start running the client's segfault handler, if it has one. In particular we can't finish the current basic block and then deliver the signal at some convenient future point, because signals like SIGILL, SIGSEGV and SIGBUS mean that the faulting insn should not simply be re-tried. (I'm sure there is a clearer way to explain this).

1.2.12. Lazy updates of the simulated program counter

Simulated `%EIP` is not updated after every simulated x86 insn as this was regarded as too expensive. Instead ucode `INCEIP` insns move it along as and when necessary. Currently we don't allow it to fall more than 4 bytes behind reality (see `VG_(disBB)` for the way this works).

Note that `%EIP` is always brought up to date by the inner dispatch loop in `VG_(dispatch)`, so that if the client takes a fault we know at least which basic block this happened in.

1.2.13. Signals

Horrible, horrible. `vg_signals.c`. Basically, since we have to intercept all system calls anyway, we can see when the client tries to install a signal handler. If it does so, we make a note of what the client asked to happen, and ask the kernel to route the signal to our own signal handler, `VG_(oursignalhandler)`. This simply notes the delivery of signals, and returns.

Every 1000 basic blocks, we see if more signals have arrived. If so, `VG_(deliver_signals)` builds signal delivery frames on the client's stack, and allows their handlers to be run. Valgrind places in these signal delivery frames a bogus return address, `VG_(signalreturn_bogusRA)`, and checks all jumps to see if any jump to it. If so, this is a sign that a signal handler is returning, and if so Valgrind removes the relevant signal frame from the client's stack, restores the from the signal frame the simulated state before the signal was delivered, and allows the client to run onwards. We have to do it this way because some signal handlers never return, they just `longjmp()`, which nukes the signal delivery frame.

The Linux kernel has a different but equally horrible hack for detecting signal handler returns. Discovering it is left as an exercise for the reader.

1.2.14. To be written

The following is a list of as-yet-not-written stuff. Apologies.

1. The translation cache and translation table
2. Exceptions, creating new translations
3. Self-modifying code

4. Errors, error contexts, error reporting, suppressions
5. Client malloc/free
6. Low-level memory management
7. A and V bitmaps
8. Symbol table management
9. Dealing with system calls
10. Namespace management
11. GDB attaching
12. Non-dependence on glibc or anything else
13. The leak detector
14. Performance problems
15. Continuous sanity checking
16. Tracing, or not tracing, child processes
17. Assembly glue for syscalls

1.3. Extensions

Some comments about Stuff To Do.

1.3.1. Bugs

Stephan Kulow and Marc Mutz report problems with kmail in KDE 3 CVS (RC2 ish) when run on Valgrind. Stephan has it deadlocking; Marc has it looping at startup. I can't repro either behaviour. Needs repro-ing and fixing.

1.3.2. Threads

Doing a good job of thread support strikes me as almost a research-level problem. The central issues are how to do fast cheap locking of the `VG_(primary_map)` structure, whether or not accesses to the individual secondary maps need locking, what race-condition issues result, and whether the already-nasty mess that is the signal simulator needs further hackery.

I realise that threads are the most-frequently-requested feature, and I am thinking about it all. If you have guru-level understanding of fast mutual exclusion mechanisms and race conditions, I would be interested in hearing from you.

1.3.3. Verification suite

Directory `tests/` contains various ad-hoc tests for Valgrind. However, there is no systematic verification or regression suite, that, for example, exercises all the stuff in `vg_memory.c`, to ensure that illegal memory accesses and undefined value uses are detected as they should be. It would be good to have such a suite.

1.3.4. Porting to other platforms

It would be great if Valgrind was ported to FreeBSD and x86 NetBSD, and to x86 OpenBSD, if it's possible (doesn't OpenBSD use a.out-style executables, not ELF ?)

The main difficulties, for an x86-ELF platform, seem to be:

- You'd need to rewrite the `/proc/self/maps` parser (`vg_procselfmaps.c`). Easy.
- You'd need to rewrite `vg_syscall_mem.c`, or, more specifically, provide one for your OS. This is tedious, but you can implement syscalls on demand, and the Linux kernel interface is, for the most part, going to look very similar to the *BSD interfaces, so it's really a copy-paste-and-modify-on-demand job. As part of this, you'd need to supply a new `vg_kerneliface.h` file.
- You'd also need to change the syscall wrappers for Valgrind's internal use, in `vg_mylibc.c`.

All in all, I think a port to x86-ELF *BSDs is not really very difficult, and in some ways I would like to see it happen, because that would force a more clear factoring of Valgrind into platform dependent and independent pieces. Not to mention, *BSD folks also deserve to use Valgrind just as much as the Linux crew do.

1.4. Easy stuff which ought to be done

1.4.1. MMX Instructions

MMX insns should be supported, using the same trick as for FPU insns. If the MMX registers are not used to copy uninitialised junk from one place to another in memory, this means we don't have to actually simulate the internal MMX unit state, so the FPU hack applies. This should be fairly easy.

1.4.2. Fix stabs-info reader

The machinery in `vg_symtab2.c` which reads "stabs" style debugging info is pretty weak. It usually correctly translates simulated program counter values into line numbers and procedure names, but the file name is often completely wrong. I think the logic used to parse "stabs" entries is weak. It should be fixed. The simplest solution, IMO, is to copy either the logic or simply the code out of GNU binutils which does this; since GDB can clearly get it right, binutils (or GDB?) must have code to do this somewhere.

1.4.3. BT/BTC/BTS/BTR

These are x86 instructions which test, complement, set, or reset, a single bit in a word. At the moment they are both incorrectly implemented and incorrectly instrumented.

The incorrect instrumentation is due to use of helper functions. This means we lose bit-level definedness tracking, which could wind up giving spurious uninitialised-value use errors. The Right Thing to do is to invent a couple of new UOpcodes, I think `GET_BIT` and `SET_BIT`, which can be used to implement all 4 x86 insns, get rid of the helpers, and give bit-accurate instrumentation rules for the two new UOpcodes.

I realised the other day that they are mis-implemented too. The x86 insns take a bit-index and a register or memory location to access. For registers the bit index clearly can only be in the range zero to register-width minus 1, and I assumed the same applied to memory locations too. But evidently not; for memory locations the index can be arbitrary, and the processor will index arbitrarily into memory as a result. This too should be fixed. Sigh. Presumably indexing outside the immediate word is not actually used by any programs yet tested on Valgrind, for otherwise they (presumably) would simply not work at all. If you plan to hack on this, first check the Intel docs to make sure my understanding is really correct.

1.4.4. Using PREFETCH Instructions

Here's a small but potentially interesting project for performance junkies. Experiments with valgrind's code generator and optimiser(s) suggest that reducing the number of instructions executed in the translations and mem-check helpers gives disappointingly small performance improvements. Perhaps this is because performance of Valgrindified code is limited by cache misses. After all, each read in the original program now gives rise to at least three reads, one for the `VG_(primary_map)`, one of the resulting secondary, and the original. Not to mention, the instrumented translations are 13 to 14 times larger than the originals. All in all one would expect the memory system to be hammered to hell and then some.

So here's an idea. An x86 insn involving a read from memory, after instrumentation, will turn into ucode of the following form:

```
... calculate effective addr, into ta and qa ...
  TESTVL qa          -- is the addr defined?
  LOADV (ta), qloaded   -- fetch V bits for the addr
  LOAD  (ta), tloaded   -- do the original load
```

At the point where the `LOADV` is done, we know the actual address (`ta`) from which the real `LOAD` will be done. We also know that the `LOADV` will take around 20 x86 insns to do. So it seems plausible that doing a prefetch of `ta` just before the `LOADV` might just avoid a miss at the `LOAD` point, and that might be a significant performance win.

Prefetch insns are notoriously tempermental, more often than not making things worse rather than better, so this would require considerable fiddling around. It's complicated because Intels and AMDs have different prefetch insns with different semantics, so that too needs to be taken into account. As a general rule, even placing the prefetches before the `LOADV` insn is too near the `LOAD`; the ideal distance is apparently circa 200 CPU cycles. So it might be worth having another analysis/transformation pass which pushes prefetches as far back as possible, hopefully immediately after the effective address becomes available.

Doing too many prefetches is also bad because they soak up bus bandwidth / cpu resources, so some cleverness in deciding which loads to prefetch and which to not might be helpful. One can imagine not prefetching client-stack-relative (`%EBP` or `%ESP`) accesses, since the stack in general tends to show good locality anyway.

There's quite a lot of experimentation to do here, but I think it might make an interesting week's work for someone.

As of 15-ish March 2002, I've started to experiment with this, using the AMD `prefetch/prefetchw` insns.

1.4.5. User-defined Permission Ranges

This is quite a large project -- perhaps a month's hacking for a capable hacker to do a good job -- but it's potentially very interesting. The outcome would be that Valgrind could detect a whole class of bugs which it currently cannot.

The presentation falls into two pieces.

1.4.5.1. Part 1: User-defined Address-range Permission Setting

Valgrind intercepts the client's `malloc`, `free`, etc calls, watches system calls, and watches the stack pointer move. This is currently the only way it knows about which addresses are valid and which not. Sometimes the client program knows extra information about its memory areas. For example, the client could at some point know that all elements of an array are out-of-date. We would like to be able to convey to Valgrind this information that the array is now addressable-but-uninitialised, so that Valgrind can then warn if elements are used before they get new values.

What I would like are some macros like this:

```
VALGRIND_MAKE_NOACCESS(addr, len)
VALGRIND_MAKE_WRITABLE(addr, len)
VALGRIND_MAKE_READABLE(addr, len)
```

and also, to check that memory is addressible/initialised,

```
VALGRIND_CHECK_ADDRESSIBLE(addr, len)
VALGRIND_CHECK_INITIALISED(addr, len)
```

I then include in my sources a header defining these macros, rebuild my app, run under Valgrind, and get user-defined checks.

Now here’s a neat trick. It’s a nuisance to have to re-link the app with some new library which implements the above macros. So the idea is to define the macros so that the resulting executable is still completely stand-alone, and can be run without Valgrind, in which case the macros do nothing, but when run on Valgrind, the Right Thing happens. How to do this? The idea is for these macros to turn into a piece of inline assembly code, which (1) has no effect when run on the real CPU, (2) is easily spotted by Valgrind’s JITter, and (3) no sane person would ever write, which is important for avoiding false matches in (2). So here’s a suggestion:

```
VALGRIND_MAKE_NOACCESS(addr, len)
```

becomes (roughly speaking)

```
movl addr, %eax
movl len,  %ebx
movl $1,   %ecx   -- 1 describes the action; MAKE_WRITABLE might be
             -- 2, etc
rorl $13, %ecx
rorl $19, %ecx
rorl $11, %eax
rorl $21, %eax
```

The rotate sequences have no effect, and it’s unlikely they would appear for any other reason, but they define a unique byte-sequence which the JITter can easily spot. Using the operand constraints section at the end of a gcc inline-assembly statement, we can tell gcc that the assembly fragment kills `%eax`, `%ebx`, `%ecx` and the condition codes, so this fragment is made harmless when not running on Valgrind, runs quickly when not on Valgrind, and does not require any other library support.

1.4.5.2. Part 2: Using it to detect Interference between Stack Variables

Currently Valgrind cannot detect errors of the following form:

```
void fooble ( void )
{
 int a[10];
 int b[10];
 a[10] = 99;
}
```

Now imagine rewriting this as

```
void fooble ( void )
{
 int spacer0;
 int a[10];
 int spacer1;
 int b[10];
 int spacer2;
 VALGRIND_MAKE_NOACCESS(&spacer0, sizeof(int));
 VALGRIND_MAKE_NOACCESS(&spacer1, sizeof(int));
 VALGRIND_MAKE_NOACCESS(&spacer2, sizeof(int));
 a[10] = 99;
}
```

Now the invalid write is certain to hit `spacer0` or `spacer1`, so Valgrind will spot the error.

There are two complications.

1. The first is that we don't want to annotate sources by hand, so the Right Thing to do is to write a C/C++ parser, annotator, prettyprinter which does this automatically, and run it on post-CPP'd C/C++ source. See http://www.cacheprof.org for an example of a system which transparently inserts another phase into the gcc/g++ compilation route. The parser/prettyprinter is probably not as hard as it sounds; I would write it in Haskell, a powerful functional language well suited to doing symbolic computation, with which I am intimately familar. There is already a C parser written in Haskell by someone in the Haskell community, and that would probably be a good starting point.

2. The second complication is how to get rid of these `NOACCESS` records inside Valgrind when the instrumented function exits; after all, these refer to stack addresses and will make no sense whatever when some other function happens to re-use the same stack address range, probably shortly afterwards. I think I would be inclined to define a special stack-specific macro:

   ```
   VALGRIND_MAKE_NOACCESS_STACK(addr, len)
   ```

   which causes Valgrind to record the client's `%ESP` at the time it is executed. Valgrind will then watch for changes in `%ESP` and discard such records as soon as the protected area is uncovered by an increase in `%ESP`. I hesitate with this scheme only because it is potentially expensive, if there are hundreds of such records, and considering that changes in `%ESP` already require expensive messing with stack access permissions.

This is probably easier and more robust than for the instrumenter program to try and spot all exit points for the procedure and place suitable deallocation annotations there. Plus C++ procedures can bomb out at any point if they get an exception, so spotting return points at the source level just won't work at all.

Although some work, it's all eminently doable, and it would make Valgrind into an even-more-useful tool.

2. How Cachegrind works

Table of Contents

2.1. Cache profiling 32
2.2. Cost centres 32
2.3. Storing cost-centres 34
2.4. Instrumentation 35
2.5. Handling basic block retranslations 37
2.6. The cache simulation 38
2.7. Output 38
2.8. Summary of performance features 39
2.9. Annotation 39
2.10. Similar work, extensions 39

2.1. Cache profiling

[Note: this document is now very old, and a lot of its contents are out of date, and misleading.]

Valgrind is a very nice platform for doing cache profiling and other kinds of simulation, because it converts horrible x86 instructions into nice clean RISC-like UCode. For example, for cache profiling we are interested in instructions that read and write memory; in UCode there are only four instructions that do this: `LOAD`, `STORE`, `FPU_R` and `FPU_W`. By contrast, because of the x86 addressing modes, almost every instruction can read or write memory.

Most of the cache profiling machinery is in the file `vg_cachesim.c`.

These notes are a somewhat haphazard guide to how Valgrind's cache profiling works.

2.2. Cost centres

Valgrind gathers cache profiling about every instruction executed, individually. Each instruction has a **cost centre** associated with it. There are two kinds of cost centre: one for instructions that don't reference memory (`iCC`), and one for instructions that do (`idCC`):

```
typedef struct _CC {
  ULong a;
  ULong m1;
  ULong m2;
} CC;

typedef struct _iCC {
  /* word 1 */
  UChar tag;
  UChar instr_size;

  /* words 2+ */
  Addr instr_addr;
  CC I;
} iCC;

typedef struct _idCC {
  /* word 1 */
  UChar tag;
  UChar instr_size;
  UChar data_size;

  /* words 2+ */
  Addr instr_addr;
  CC I;
  CC D;
} idCC;
```

Each `CC` has three fields `a`, `m1`, `m2` for recording references, level 1 misses and level 2 misses. Each of these is a 64-bit `ULong` -- the numbers can get very large, ie. greater than 4.2 billion allowed by a 32-bit unsigned int.

A `iCC` has one `CC` for instruction cache accesses. A `idCC` has two, one for instruction cache accesses, and one for data cache accesses.

The `iCC` and `dCC` structs also store unchanging information about the instruction:

- An instruction-type identification tag (explained below)
- Instruction size
- Data reference size (`idCC` only)
- Instruction address

Note that data address is not one of the fields for `idCC`. This is because for many memory-referencing instructions the data address can change each time it's executed (eg. if it uses register-offset addressing). We have to give this item to the cache simulation in a different way (see Instrumentation section below). Some memory-referencing instructions do always reference the same address, but we don't try to treat them specialy in order to keep things simple.

Also note that there is only room for recording info about one data cache access in an `idCC`. So what about instructions that do a read then a write, such as:

```
inc %(esi)
```

In a write-allocate cache, as simulated by Valgrind, the write cannot miss, since it immediately follows the read which will drag the block into the cache if it's not already there. So the write access isn't really interesting, and Valgrind doesn't record it. This means that Valgrind doesn't measure memory references, but rather memory references that could miss in the cache. This behaviour is the same as that used by the AMD Athlon hardware counters. It also has the benefit of simplifying the implementation -- instructions that read and write memory can be treated like instructions that read memory.

2.3. Storing cost-centres

Cost centres are stored in a way that makes them very cheap to lookup, which is important since one is looked up for every original x86 instruction executed.

Valgrind does JIT translations at the basic block level, and cost centres are also setup and stored at the basic block level. By doing things carefully, we store all the cost centres for a basic block in a contiguous array, and lookup comes almost for free.

Consider this part of a basic block (for exposition purposes, pretend it's an entire basic block):

```
movl $0x0,%eax
movl $0x99, -4(%ebp)
```

The translation to UCode looks like this:

```
MOVL     $0x0, t20
PUTL     t20, %EAX
INCEIPo   $5

LEA1L    -4(t4), t14
MOVL     $0x99, t18
STL     t18, (t14)
INCEIPo   $7
```

The first step is to allocate the cost centres. This requires a preliminary pass to count how many x86 instructions were in the basic block, and their types (and thus sizes). UCode translations for single x86 instructions are delimited by the `INCEIPo` instruction, the argument of which gives the byte size of the instruction (note that lazy INCEIP updating is turned off to allow this).

We can tell if an x86 instruction references memory by looking for `LDL` and `STL` UCode instructions, and thus what kind of cost centre is required. From this we can determine how many cost centres we need for the basic block, and their sizes. We can then allocate them in a single array.

Consider the example code above. After the preliminary pass, we know we need two cost centres, one `iCC` and one `dCC`. So we allocate an array to store these which looks like this:

```
|(uninit)|    tag       (1 byte)
|(uninit)|    instr_size  (1 bytes)
|(uninit)|    (padding)   (2 bytes)
|(uninit)|    instr_addr  (4 bytes)
|(uninit)|    I.a       (8 bytes)
|(uninit)|    I.m1      (8 bytes)
|(uninit)|    I.m2      (8 bytes)

|(uninit)|    tag       (1 byte)
|(uninit)|    instr_size  (1 byte)
|(uninit)|    data_size   (1 byte)
|(uninit)|    (padding)   (1 byte)
|(uninit)|    instr_addr  (4 bytes)
|(uninit)|    I.a       (8 bytes)
|(uninit)|    I.m1      (8 bytes)
|(uninit)|    I.m2      (8 bytes)
|(uninit)|    D.a       (8 bytes)
|(uninit)|    D.m1      (8 bytes)
|(uninit)|    D.m2      (8 bytes)
```

(We can see now why we need tags to distinguish between the two types of cost centres.)

We also record the size of the array. We look up the debug info of the first instruction in the basic block, and then stick the array into a table indexed by filename and function name. This makes it easy to dump the information quickly to file at the end.

2.4. Instrumentation

The instrumentation pass has two main jobs:

1. Fill in the gaps in the allocated cost centres.
2. Add UCode to call the cache simulator for each instruction.

The instrumentation pass steps through the UCode and the cost centres in tandem. As each original x86 instruction's UCode is processed, the appropriate gaps in the instructions cost centre are filled in, for example:

```
|INSTR_CC|    tag       (1 byte)
|5      |    instr_size  (1 bytes)
|(uninit)|    (padding)   (2 bytes)
|i_addr1 |    instr_addr  (4 bytes)
|0      |    I.a        (8 bytes)
|0      |    I.m1       (8 bytes)
|0      |    I.m2       (8 bytes)

|WRITE_CC|    tag       (1 byte)
|7      |    instr_size  (1 byte)
|4      |    data_size  (1 byte)
|(uninit)|    (padding)   (1 byte)
|i_addr2 |    instr_addr  (4 bytes)
|0      |    I.a        (8 bytes)
|0      |    I.m1       (8 bytes)
|0      |    I.m2       (8 bytes)
|0      |    D.a        (8 bytes)
|0      |    D.m1       (8 bytes)
|0      |    D.m2       (8 bytes)
```

(Note that this step is not performed if a basic block is re-translated; see Handling basic block retranslations [37] for more information.)

GCC inserts padding before the `instr_size` field so that it is word aligned.

The instrumentation added to call the cache simulation function looks like this (instrumentation is indented to distinguish it from the original UCode):

```
MOVL     $0x0, t20
PUTL     t20, %EAX
  PUSHL    %eax
  PUSHL    %ecx
  PUSHL    %edx
  MOVL     $0x4091F8A4, t46  # address of 1st CC
  PUSHL    t46
  CALLMo   $0x12           # second cachesim function
  CLEARo   $0x4
  POPL     %edx
  POPL     %ecx
  POPL     %eax
INCEIPo  $5

LEA1L    -4(t4), t14
MOVL     $0x99, t18
  MOVL     t14, t42
STL      t18, (t14)
  PUSHL    %eax
  PUSHL    %ecx
  PUSHL    %edx
  PUSHL    t42
  MOVL     $0x4091F8C4, t44  # address of 2nd CC
  PUSHL    t44
  CALLMo   $0x13           # second cachesim function
  CLEARo   $0x8
  POPL     %edx
  POPL     %ecx
  POPL     %eax
INCEIPo  $7
```

Consider the first instruction's UCode. Each call is surrounded by three `PUSHL` and `POPL` instructions to save and restore the caller-save registers. Then the address of the instruction's cost centre is pushed onto the stack, to be the first argument to the cache simulation function. The address is known at this point because we are doing a simultaneous pass through the cost centre array. This means the cost centre lookup for each instruction is almost free (just the cost of pushing an argument for a function call). Then the call to the cache simulation function for non-memory-reference instructions is made (note that the `CALLMo` UInstruction takes an offset into a table of predefined functions; it is not an absolute address), and the single argument is `CLEAR`ed from the stack.

The second instruction's UCode is similar. The only difference is that, as mentioned before, we have to pass the address of the data item referenced to the cache simulation function too. This explains the `MOVL t14, t42` and `PUSHL t42` UInstructions. (Note that the seemingly redundant `MOV`ing will probably be optimised away during register allocation.)

Note that instead of storing unchanging information about each instruction (instruction size, data size, etc) in its cost centre, we could have passed in these arguments to the simulation function. But this would slow the calls down (two or three extra arguments pushed onto the stack). Also it would bloat the UCode instrumentation by amounts similar to the space required for them in the cost centre; bloated UCode would also fill the translation cache more quickly, requiring more translations for large programs and slowing them down more.

2.5. Handling basic block retranslations

The above description ignores one complication. Valgrind has a limited size cache for basic block translations; if it fills up, old translations are discarded. If a discarded basic block is executed again, it must be re-translated.

However, we can't use this approach for profiling -- we can't throw away cost centres for instructions in the middle of execution! So when a basic block is translated, we first look for its cost centre array in the hash table. If there is no cost centre array, it must be the first translation, so we proceed as described above. But if there is a cost centre array already, it must be a retranslation. In this case, we skip the cost centre allocation and initialisation steps, but still do the UCode instrumentation step.

2.6. The cache simulation

The cache simulation is fairly straightforward. It just tracks which memory blocks are in the cache at the moment (it doesn't track the contents, since that is irrelevant).

The interface to the simulation is quite clean. The functions called from the UCode contain calls to the simulation functions in the files `vg_cachesim_{I1,D1,L2}.c`; these calls are inlined so that only one function call is done per simulated x86 instruction. The file `vg_cachesim.c` simply `#include`s the three files containing the simulation, which makes plugging in new cache simulations is very easy -- you just replace the three files and recompile.

2.7. Output

Output is fairly straightforward, basically printing the cost centre for every instruction, grouped by files and functions. Total counts (eg. total cache accesses, total L1 misses) are calculated when traversing this structure rather than during execution, to save time; the cache simulation functions are called so often that even one or two extra adds can make a sizeable difference.

Input file has the following 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=" | "fi=" | "fe=") filename
fn_line      ::= "fn=" fn_name
count_line   ::= line_num ws? (count ws)+
summary_line ::= "summary:" ws? (count ws)+
count        ::= num | "."
```

Where:

- `non_nl_string` is any string not containing a newline.
- `cmd` is a command line invocation.
- `filename` and `fn_name` can be anything.
- `num` and `line_num` are decimal numbers.
- `ws` is whitespace.

- `nl` is a newline.

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

Counts can be "." to represent "N/A", eg. the number of write misses for an instruction that doesn't write to memory.

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 "." entry.

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_line`s to give the context of the first `count_line`s.

Each `file_line` should be immediately followed by a `fn_line`. "fi=" `file_lines` are used to switch filenames for inlined functions; "fe=" `file_lines` are similar, but are put at the end of a basic block in which the file name hasn't been switched back to the original file name. (fi and fe lines behave the same, they are only distinguished to help debugging.)

2.8. Summary of performance features

Quite a lot of work has gone into making the profiling as fast as possible. This is a summary of the important features:

- The basic block-level cost centre storage allows almost free cost centre lookup.
- Only one function call is made per instruction simulated; even this accounts for a sizeable percentage of execution time, but it seems unavoidable if we want flexibility in the cache simulator.
- Unchanging information about an instruction is stored in its cost centre, avoiding unnecessary argument pushing, and minimising UCode instrumentation bloat.
- Summary counts are calculated at the end, rather than during execution.
- The `cachegrind.out` output files can contain huge amounts of information; file format was carefully chosen to minimise file sizes.

2.9. Annotation

Annotation is done by cg_annotate. It is a fairly straightforward Perl script that slurps up all the cost centres, and then runs through all the chosen source files, printing out cost centres with them. It too has been carefully optimised.

2.10. Similar work, extensions

It would be relatively straightforward to do other simulations and obtain line-by-line information about interesting events. A good example would be branch prediction -- all branches could be instrumented to interact with a branch prediction simulator, using very similar techniques to those described above.

In particular, cg_annotate would not need to change -- the file format is such that it is not specific to the cache simulation, but could be used for any kind of line-by-line information. The only part of cg_annotate that is specific to the cache simulation is the name of the input file (`cachegrind.out`), although it would be very simple to add an option to control this.

3. Writing a New Valgrind Tool

Table of Contents

3.1. Introduction 40
  3.1.1. Supervised Execution 40
  3.1.2. Tools 40
  3.1.3. Execution Spaces 41
3.2. Writing a Tool 41
  3.2.1. Why write a tool? 41
  3.2.2. Suggested tools 42
  3.2.3. How tools work 43
  3.2.4. Getting the code 43
  3.2.5. Getting started 43
  3.2.6. Writing the code 45
  3.2.7. Initialisation 45
  3.2.8. Instrumentation 45
  3.2.9. Finalisation 46
  3.2.10. Other Important Information 46
  3.2.11. Words of Advice 46
3.3. Advanced Topics 47
  3.3.1. Suppressions 47
  3.3.2. Documentation 47
  3.3.3. Regression Tests 49
  3.3.4. Profiling 50
  3.3.5. Other Makefile Hackery 50
  3.3.6. Core/tool Interface Versions 50
3.4. Final Words 50

3.1. Introduction

3.1.1. Supervised Execution

Valgrind provides a generic infrastructure for supervising the execution of programs. This is done by providing a way to instrument programs in very precise ways, making it relatively easy to support activities such as dynamic error detection and profiling.

Although writing a tool is not easy, and requires learning quite a few things about Valgrind, it is much easier than instrumenting a program from scratch yourself.

[Nb: What follows is slightly out of date. In particular, there are various references to a file include/tool.h which has been split into a number of header files in include/.]

3.1.2. Tools

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 x86-to-x86 JIT compiler, low-level memory manager, signal handling and a scheduler (for pthreads). It also provides certain services that are useful to some but not all tools, such as support for error recording and suppression.

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 define certain variables 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.

3.1.3. Execution Spaces

An important concept to understand before writing a tool is that there are three spaces in which program code executes:

1. User space: this covers most of the program's execution. The tool is given the code and can instrument it any way it likes, providing (more or less) total control over the code.

   Code executed in user space includes all the program code, almost all of the C library (including things like the dynamic linker), and almost all parts of all other libraries.

2. Core space: a small proportion of the program's execution takes place entirely within Valgrind's core. This includes:

   - Dynamic memory management (`malloc()` etc.)
   - Pthread operations and scheduling
   - Signal handling

   A tool has no control over these operations; it never "sees" the code doing this work and thus cannot instrument it. However, the core provides hooks so a tool can be notified when certain interesting events happen, for example when when dynamic memory is allocated or freed, the stack pointer is changed, or a pthread mutex is locked, etc.

   Note that these hooks only notify tools of events relevant to user space. For example, when the core allocates some memory for its own use, the tool is not notified of this, because it's not directly part of the supervised program's execution.

3. Kernel space: execution in the kernel. Two kinds:

   a. System calls: can't be directly observed by either the tool or the core. But the core does have some idea of what happens to the arguments, and it provides hooks for a tool to wrap system calls.

   b. Other: all other kernel activity (e.g. process scheduling) is totally opaque and irrelevant to the program.

4. It should be noted that a tool only has direct control over code executed in user space. This is the vast majority of code executed, but it is not absolutely all of it, so any profiling information recorded by a tool won't be totally accurate.

3.2. Writing a Tool

3.2.1. Why write a tool?

Before you write a tool, you should have some idea of what it should do. What is it you want to know about your programs of interest? Consider some existing tools:

- **memcheck**: among other things, performs fine-grained validity and addressibility checks of every memory reference performed by the program.
- **addrcheck**: performs lighterweight addressibility checks of every memory reference performed by the program.
- **cachegrind**: tracks every instruction and memory reference to simulate instruction and data caches, tracking cache accesses and misses that occur on every line in the program.
- **helgrind**: tracks every memory access and mutex lock/unlock to determine if a program contains any data races.
- **lackey**: does simple counting of various things: the number of calls to a particular function (`_dl_runtime_resolve()`); the number of basic blocks, x86 instruction, UCode instructions executed; the number of branches executed and the proportion of those which were taken.

These examples give a reasonable idea of what kinds of things Valgrind can be used for. The instrumentation can range from very lightweight (e.g. counting the number of times a particular function is called) to very intrusive (e.g. memcheck's memory checking).

3.2.2. Suggested tools

Here is a list of ideas we have had for tools that should not be too hard to implement.

- **branch profiler**: A machine's branch prediction hardware could be simulated, and each branch annotated with the number of predicted and mispredicted branches. Would be implemented quite similarly to Cachegrind, and could reuse the `cg_annotate` script to annotate source code.

  The biggest difficulty with this is the simulation; the chip-makers are very cagey about how their chips do branch prediction. But implementing one or more of the basic algorithms could still give good information.

- **coverage tool**: Cachegrind can already be used for doing test coverage, but it's massive overkill to use it just for that.

  It would be easy to write a coverage tool that records how many times each basic block was recorded. Again, the `cg_annotate` script could be used for annotating source code with the gathered information. Although, `cg_annotate` is only designed for working with single program runs. It could be extended relatively easily to deal with multiple runs of a program, so that the coverage of a whole test suite could be determined.

  In addition to the standard coverage information, such a tool could record extra information that would help a user generate test cases to exercise unexercised paths. For example, for each conditional branch, the tool could record all inputs to the conditional test, and print these out when annotating.

- **run-time type checking**: A nice example of a dynamic checker is given in this paper:

  Debugging via Run-Time Type Checking
  Alexey Loginov, Suan Hsi Yong, Susan Horwitz and Thomas Reps
  Proceedings of Fundamental Approaches to Software Engineering
  April 2001.

  Similar is the tool described in this paper:

  Run-Time Type Checking for Binary Programs
  Michael Burrows, Stephen N. Freund, Janet L. Wiener
  Proceedings of the 12th International Conference on Compiler Construction (CC 2003)
  April 2003.

  This approach can find quite a range of bugs, particularly in C and C++ programs, and could be implemented quite nicely as a Valgrind tool.

  Ways to speed up this run-time type checking are described in this paper:

  Reducing the Overhead of Dynamic Analysis
  Suan Hsi Yong and Susan Horwitz
  Proceedings of Runtime Verification ’02
  July 2002.

  Valgrind’s client requests could be used to pass information to a tool about which elements need instrumentation and which don’t.

We would love to hear from anyone who implements these or other tools.

3.2.3. How tools work

Tools must define various functions for instrumenting programs that are called by Valgrind’s core, yet they must be implemented in such a way that they can be written and compiled without touching Valgrind’s core. This is important, because one of our aims is to allow people to write and distribute their own tools that can be plugged into Valgrind’s core easily.

This is achieved by packaging each tool into a separate shared object which is then loaded ahead of the core shared object `valgrind.so`, using the dynamic linker’s `LD_PRELOAD` variable. Any functions defined in the tool that share the name with a function defined in core (such as the instrumentation function `instrument()`) override the core’s definition. Thus the core can call the necessary tool functions.

This magic is all done for you; the shared object used is chosen with the `--tool` option to the `valgrind` startup script. The default tool used is `memcheck`, Valgrind’s original memory checker.

3.2.4. Getting the code

To write your own tool, you’ll need the Valgrind source code. A normal source distribution should do, although you might want to check out the latest code from the Subversion repository. See the information about how to do so at the Valgrind website [http://www.valgrind.org/].

3.2.5. 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 an abbreviation that can be used as a short prefix. We'll use `foobar` and `fb` as an example.
2. Make a new directory `foobar/` which will hold the tool.
3. Copy `none/Makefile.am` into `foobar/`. Edit it by replacing all occurrences of the string `"none"` with `"foobar"` and the one occurrence of the string `"nl_"` with `"fb_"`. It might be worth trying to understand this file, at least a little; you might have to do more complicated things with it later on. In particular, the name of the `vgtool_foobar_so_SOURCES` variable determines the name of the tool's shared object, which determines what name must be passed to the `--tool` option to use the tool.
4. Copy `none/nl_main.c` into `foobar/`, renaming it as `fb_main.c`. Edit it by changing the lines in `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.
5. Edit `Makefile.am`, adding the new directory `foobar` to the `SUBDIRS` variable.
6. Edit `configure.in`, adding `foobar/Makefile` to the `AC_OUTPUT` list.
7. Run:

   ```
   autogen.sh
   ./configure --prefix=‘pwd‘/inst
   make install
   ```

   It should automake, configure and compile without errors, putting copies of the tool's shared object `vgtool_foobar.so` in `foobar/` and `inst/lib/valgrind/`.
8. 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 for x86-linux.
   ==738== Copyright (C) 1066AD, and GNU GPL’d, by J. Random Hacker.
   ==738== Built with valgrind-1.1.0, a program execution monitor.
   ==738== Copyright (C) 2000-2003, and GNU GPL’d, by Julian Seward.
   ==738== Estimated CPU clock rate is 1400 MHz
   ==738== For more details, rerun with: -v
   ==738== Wed Sep 25 10:31:54 BST 2002
   ==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...

3.2.6. Writing the code

A tool must define at least these four functions:

```
pre_clo_init()
post_clo_init()
instrument()
fini()
```

Also, it must use the macro `VG_DETERMINE_INTERFACE_VERSION` exactly once in its source code. If it doesn't, you will get a link error involving `VG_(tool_interface_version)`. This macro is used to ensure the core/tool interface used by the core and a plugged-in tool are binary compatible.

In addition, if a tool wants to use some of the optional services provided by the core, it may have to define other functions.

3.2.7. 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 malloc'd blocks, etc.

For example, if a tool wants the core's help in recording and reporting errors, it must set the `tool_errors` need to `True`, and then provide definitions of six 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. See the type `VgNeeds` in `include/tool.h` for full details of all the needs.

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 blocks of memory being malloc'd, the stack pointer changing, a mutex being locked, etc. If a tool wants to know about this, it should set the relevant pointer in the structure to point to a function, which will be called when that event happens.

For example, if the tool want to be notified when a new block of memory is malloc'd, 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/tool.h`.

3.2.8. Instrumentation

`instrument()` is the interesting one. It allows you to instrument *UCode*, which is Valgrind's RISC-like intermediate language. UCode is described in Introduction to UCode [12].

The easiest way to instrument UCode 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.

A much more complicated way to instrument UCode, albeit one that might result in faster instrumented programs, is to extend UCode with new UCode instructions. This is recommended for advanced Valgrind hackers only! See Memcheck for an example.

3.2.9. 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.

3.2.10. 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 file `include/tool.h` contains all the types, macros, functions, etc. that a tool should (hopefully) need, and is the only `.h` file a tool should need to `#include`.

In particular, you probably shouldn'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 `tool.h`.

Similarly, when writing a tool, you shouldn't need to look at any of the code in Valgrind's core. Although it might be useful sometimes to help understand something.

`tool.h` has a reasonable amount of documentation in it that should hopefully be enough to get you going. But ultimately, the tools distributed (Memcheck, Addrcheck, Cachegrind, Lackey, etc.) are probably the best documentation of all, for the moment.

Note that the `VG_` and `TL_` macros are used heavily. These just prepend longer strings in front of names to avoid potential namespace clashes. We strongly recommend using the `TL_` macro for any global functions and variables in your tool, or writing a similar macro.

3.2.11. Words of Advice

Writing and debugging tools is not trivial. Here are some suggestions for solving common problems.

3.2.11.1. Segmentation Faults

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.

3.2.11.2. Debugging C functions

If you want to debug C functions used by your tool, you can attach GDB to Valgrind with some effort:

1. Enable the following code in `coregrind/vg_main.c` by changing `if (0)` into `if (1)`:

   ```
   /* Hook to delay things long enough so we can get the pid and
      attach GDB in another shell. */
   if (0) {
     Int p, q;
     for ( p = 0; p < 50000; p++ )
       for ( q = 0; q < 50000; q++ ) ;
   }
   ```

   and rebuild Valgrind.

2. Then run:

   ```
   valgrind <prog>
   ```

   Valgrind starts the program, printing its process id, and then delays for a few seconds (you may have to change the loop bounds to get a suitable delay).

3. In a second shell run:

   ```
   gdb <prog pid>
   ```

GDB may be able to give you useful information. Note that by default most of the system is built with `-fomit-frame-pointer`, and you'll need to get rid of this to extract useful tracebacks from GDB.

3.2.11.3. UCode Instrumentation Problems

If you are having problems with your UCode instrumentation, it's likely that GDB won't be able to help at all. In this case, Valgrind's `--trace-codegen` option is invaluable for observing the results of instrumentation.

3.2.11.4. Miscellaneous

If you just want to know whether a program point has been reached, using the `OINK` macro (in `include/tool.h`) can be easier than using GDB.

The other debugging command line options can be useful too (run `valgrind -h` for the list).

3.3. Advanced Topics

Once a tool becomes more complicated, there are some extra things you may want/need to do.

3.3.1. 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 `valgrind/*.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)()`.

3.3.2. Documentation

As of version 3.0.1, Valgrind documentation has been converted to XML. Why? See The XML FAQ [http://www.ucc.ie/xml/].

3.3.2.1. The XML Toolchain

If you are feeling conscientious and want to write some documentation for your tool, please use XML. The Valgrind Docs use the following toolchain and versions:

```
xmllint:  using libxml version 20607
xsltproc:  using libxml 20607, libxslt 10102 and libexslt 802
pdfxmltex: pdfTeX (Web2C 7.4.5) 3.14159-1.10b
pdftops:  version 3.00
DocBook:  version 4.2
```

Latency: you should note that latency is a big problem: DocBook is constantly being updated, but the tools tend to lag behind somewhat. It is important that the versions get on with each other, so if you decide to upgrade something, then you need to ascertain whether things still work nicely - this *cannot* be assumed.

Stylesheets: The Valgrind docs use various custom stylesheet layers, all of which are in `valgrind/docs/lib/`. You shouldn’t need to modify these in any way.

Catalogs: Assuming that you have the various tools listed above installed, you will probably need to modify `valgrind/docs/lib/vg-catalog.xml` so that the parser can find your DocBook installation. Catalogs provide a mapping from generic addresses to specific local directories on a given machine. Just add another `group` to this file, reflecting your local installation.

3.3.2.2. Writing the Documentation

Follow these steps (using `foobar` as the example tool name again):

1. Make a directory `valgrind/foobar/docs/`.
2. Copy the XML documentation file for the tool Nulgrind from `valgrind/none/docs/nl-manual.xml` to `foobar/docs/`, and rename it to `foobar/docs/fb-manual.xml`.

   Note: there is a *really stupid* tetex bug with underscores in filenames, so don’t use ’`_`’.
3. Write the documentation. There are some helpful bits and pieces on using xml markup in `valgrind/docs/xml/xml_help.txt`.
4. Include it in the User Manual by adding the relevant entry must be added to `valgrind/docs/xml/manual.xml`. Copy and edit an existing entry.

5. Validate `foobar/docs/fb-manual.xml` using the following command from within `valgrind/docs/`:

   ```
   % make valid
   ```

   You will probably 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.

6. 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 `valgrind/docs/html/`. Use the following command, within `valgrind/docs/`:

   ```
   % make html-docs
   ```

7. When you have finished, also generate pdf and ps output to check all is well, from within `valgrind/docs/`:

   ```
   % make print-docs
   ```

   Check the output `.pdf` and `.ps` files in `valgrind/docs/print/`.

3.3.3. Regression Tests

Valgrind has some support for regression tests. If you want to write regression tests for your tool:

1. Make a directory `foobar/tests/`.
2. Edit `foobar/Makefile.am`, adding `tests` to the `SUBDIRS` variable.
3. Edit `configure.in`, adding `foobar/tests/Makefile` to the `AC_OUTPUT` list.
4. Write `foobar/tests/Makefile.am`. Use `memcheck/tests/Makefile.am` as an example.
5. 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`.
6. 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.

3.3.4. Profiling

Nb: as of 25-Mar-2005, the profiling is broken, and has been for a long time...

To do simple tick-based profiling of a tool, include the line:

```
#include "vg_profile.c"
```

in the tool somewhere, and rebuild (you may have to `make clean` first). Then run Valgrind with the `--profile=yes` option.

The profiler is stack-based; you can register a profiling event with `VG_(register_profile_event)()` and then use the `VGP_PUSHCC` and `VGP_POPCC` macros to record time spent doing certain things. New profiling event numbers must not overlap with the core profiling event numbers. See `include/tool.h` for details and Memcheck for an example.

3.3.5. Other Makefile Hackery

If you add any directories under `valgrind/foobar/`, you will need to add an appropriate `Makefile.am` to it, and add a corresponding entry to the `AC_OUTPUT` list in `valgrind/configure.in`.

If you add any scripts to your tool (see Cachegrind for an example) you need to add them to the `bin_SCRIPTS` variable in `valgrind/foobar/Makefile.am`.

3.3.6. Core/tool Interface Versions

In order to allow for the core/tool interface to evolve over time, Valgrind uses a basic interface versioning system. All a tool has to do is use the `VG_DETERMINE_INTERFACE_VERSION` macro exactly once in its code. If not, a link error will occur when the tool is built.

The interface version number has the form X.Y. Changes in Y indicate binary compatible changes. Changes in X indicate binary incompatible changes. If the core and tool has the same major version number X they should work together. If X doesn’t match, Valgrind will abort execution with an explanation of the problem.

This approach was chosen so that if the interface changes in the future, old tools won’t work and the reason will be clearly explained, instead of possibly crashing mysteriously. We have attempted to minimise the potential for binary incompatible changes by means such as minimising the use of naked structs in the interface.

3.4. Final Words

This whole core/tool business is under active development, although it’s slowly maturing.

The first consequence of this is that the core/tool interface will continue to change in the future; we have no intention of freezing it and then regretting the inevitable stupidities. Hopefully most of the future changes will be to add new features, hooks, functions, etc, rather than to change old ones, which should cause a minimum of trouble for existing tools, and we’ve put some effort into future-proofing the interface to avoid binary incompatibility. But we can’t guarantee anything. The versioning system should catch any incompatibilities. Just something to be aware of.

The second consequence of this is that we’d love to hear your feedback about it:

- If you love it or hate it

- If you find bugs
- If you write a tool
- If you have suggestions for new features, needs, trackable events, functions
- If you have suggestions for making tools easier to write
- If you have suggestions for improving this documentation
- If you don’t understand something

or anything else!

Happy programming.

Distribution Documents

Distribution Documents

Table of Contents

1. ACKNOWLEDGEMENTS 4
2. AUTHORS 6
3. INSTALL 7
4. NEWS 11
5. README 32
6. README_MISSING_SYSCALL_OR_IOCTL 34
7. README_DEVELOPERS 38
8. README_PACKAGERS 40

1. ACKNOWLEDGEMENTS

Cerion Armour-Brown, cerion@open-works.co.uk

Cerion worked on PowerPC instruction set support using the Vex
dynamic-translation framework.

Jeremy Fitzhardinge, jeremy@valgrind.org

Jeremy wrote Helgrind and totally overhauled low-level syscall/signal
and address space layout stuff, among many other improvements.

Tom Hughes, tom@valgrind.org

Tom did a vast number of bug fixes, and helped out with support for
more recent Linux/glibc versions.

Nicholas Nethercote, njn@valgrind.org

Nick did the core/tool generalisation, wrote Cachegrind and Massif,
and tons of other stuff.

Paul Mackerras

Paul did a lot of the initial per-architecture factoring that forms
the basis of the 3.0 line and is also to be 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.

Dirk Mueller, dmuell@gmx.net

Dirk contributed the malloc-free mismatch checking stuff and various
other bits and pieces, and acted as our KDE liaison.

Donna Robinson, donna@terpsichore.ws

Keeper of the very excellent http://www.valgrind.org.

Julian Seward, julian@valgrind.org

Julian was the original designer and author of Valgrind, created the
dynamic translation framework, wrote Memcheck and Addrcheck, and did
lots of other things.

Robert Walsh, rjwalsh@valgrind.org

Robert added file descriptor leakage checking, new library
interception machinery, support for client allocation pools, and minor
other tweakage.

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.

And lots and lots of other people sent bug reports, patches, and very helpful feedback.

2. AUTHORS

Cerion Armour-Brown worked on PowerPC instruction set support using
the Vex dynamic-translation framework.

Jeremy Fitzhardinge wrote Helgrind and totally overhauled low-level
syscall/signal and address space layout stuff, among many other things.

Tom Hughes did a vast number of bug fixes, and helped out with support
for more recent Linux/glibc versions.

Nicholas Nethercote did the core/tool generalisation, wrote
Cachegrind and Massif, and tons of other stuff.

Paul Mackerras did a lot of the initial per-architecture factoring
that forms the basis of the 3.0 line and is also to be 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.

Dirk Mueller contributed the malloc-free mismatch checking stuff
and other bits and pieces, and acted as our KDE liaison.

Julian Seward was the original founder, designer and author, created
the dynamic translation frameworks, wrote Memcheck and Addrcheck, and
did lots of other things.

Robert Walsh added file descriptor leakage checking, new library
interception machinery, support for client allocation pools, and minor
other tweakage.

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.

And lots and lots of other people sent bug reports, patches, and very
helpful feedback. Thank you all.

3. INSTALL

Basic Installation
==================

These are generic installation instructions.

The ‘configure’ shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a ‘Makefile’ in each directory of the package. It may also create one or more ‘.h’ files containing system-dependent definitions. Finally, it creates a shell script ‘config.status’ that you can run in the future to recreate the current configuration, a file ‘config.cache’ that saves the results of its tests to speed up reconfiguring, and a file ‘config.log’ containing compiler output (useful mainly for debugging ‘configure’).

If you need to do unusual things to compile the package, please try to figure out how ‘configure’ could check whether to do them, and mail diffs or instructions to the address given in the ‘README’ so they can be considered for the next release. If at some point ‘config.cache’ contains results you don’t want to keep, you may remove or edit it.

The file ‘configure.in’ is used to create ‘configure’ by a program called ‘autoconf’. You only need ‘configure.in’ if you want to change it or regenerate ‘configure’ using a newer version of ‘autoconf’.

The simplest way to compile this package is:

1. ‘cd’ to the directory containing the package’s source code and type ‘./configure’ to configure the package for your system. If you’re using ‘csh’ on an old version of System V, you might need to type ‘sh ./configure’ instead to prevent ‘csh’ from trying to execute ‘configure’ itself.

   Running ‘configure’ takes awhile. While running, it prints some messages telling which features it is checking for.

2. Type ‘make’ to compile the package.

3. Optionally, type ‘make check’ to run any self-tests that come with the package.

4. Type ‘make install’ to install the programs and any data files and documentation.

5. You can remove the program binaries and object files from the source code directory by typing ‘make clean’. To also remove the files that ‘configure’ created (so you can compile the package for a different kind of computer), type ‘make distclean’. There is also a ‘make maintainer-clean’ target, but that is intended mainly for the package’s developers. If you use it, you may have to get

all sorts of other programs in order to regenerate files that came with the distribution.

Compilers and Options
=====================

Some systems require unusual options for compilation or linking that the ‘configure’ script does not know about. You can give ‘configure’ initial values for variables by setting them in the environment. Using a Bourne-compatible shell, you can do that on the command line like this:

```
CC=c89 CFLAGS=-O2 LIBS=-lposix ./configure
```

Or on systems that have the ‘env’ program, you can do it like this:

```
env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./configure
```

Compiling For Multiple Architectures
====================================

You can compile the package for more than one kind of computer at the same time, by placing the object files for each architecture in their own directory. To do this, you must use a version of ‘make’ that supports the ‘VPATH’ variable, such as GNU ‘make’. ‘cd’ to the directory where you want the object files and executables to go and run the ‘configure’ script. ‘configure’ automatically checks for the source code in the directory that ‘configure’ is in and in ‘..’.

If you have to use a ‘make’ that does not supports the ‘VPATH’ variable, you have to compile the package for one architecture at a time in the source code directory. After you have installed the package for one architecture, use ‘make distclean’ before reconfiguring for another architecture.

Installation Names
==================

By default, ‘make install’ will install the package’s files in ‘/usr/local/bin’, ‘/usr/local/man’, etc. You can specify an installation prefix other than ‘/usr/local’ by giving ‘configure’ the option ‘--prefix=PATH’.

You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If you give ‘configure’ the option ‘--exec-prefix=PATH’, the package will use PATH as the prefix for installing programs and libraries. Documentation and other data files will still use the regular prefix.

In addition, if you use an unusual directory layout you can give options like ‘--bindir=PATH’ to specify different values for particular kinds of files. Run ‘configure --help’ for a list of the directories you can set and what kinds of files go in them.

If the package supports it, you can cause programs to be installed with an extra prefix or suffix on their names by giving ‘configure’ the option ‘--program-prefix=PREFIX’ or ‘--program-suffix=SUFFIX’.

Optional Features
=================

Some packages pay attention to ‘--enable-FEATURE’ options to ‘configure’, where FEATURE indicates an optional part of the package. They may also pay attention to ‘--with-PACKAGE’ options, where PACKAGE is something like ‘gnu-as’ or ‘x’ (for the X Window System). The ‘README’ should mention any ‘--enable-’ and ‘--with-’ options that the package recognizes.

For packages that use the X Window System, ‘configure’ can usually find the X include and library files automatically, but if it doesn’t, you can use the ‘configure’ options ‘--x-includes=DIR’ and ‘--x-libraries=DIR’ to specify their locations.

Specifying the System Type
==========================

There may be some features ‘configure’ can not figure out automatically, but needs to determine by the type of host the package will run on. Usually ‘configure’ can figure that out, but if it prints a message saying it can not guess the host type, give it the ‘--host=TYPE’ option. TYPE can either be a short name for the system type, such as ‘sun4’, or a canonical name with three fields:

CPU-COMPANY-SYSTEM

See the file ‘config.sub’ for the possible values of each field. If ‘config.sub’ isn’t included in this package, then this package doesn’t need to know the host type.

If you are building compiler tools for cross-compiling, you can also use the ‘--target=TYPE’ option to select the type of system they will produce code for and the ‘--build=TYPE’ option to select the type of system on which you are compiling the package.

Sharing Defaults
================

If you want to set default values for ‘configure’ scripts to share, you can create a site shell script called ‘config.site’ that gives default values for variables like ‘CC’, ‘cache_file’, and ‘prefix’. ‘configure’ looks for ‘PREFIX/share/config.site’ if it exists, then ‘PREFIX/etc/config.site’ if it exists. Or, you can set the ‘CONFIG_SITE’ environment variable to the location of the site script. A warning: not all ‘configure’ scripts look for a site script.

Operation Controls
==================

‘configure’ recognizes the following options to control how it operates.

'--cache-file=FILE'
: Use and save the results of the tests in FILE instead of './config.cache'. Set FILE to '/dev/null' to disable caching, for debugging 'configure'.

'--help'
: Print a summary of the options to 'configure', and exit.

'--quiet'
'--silent'
'-q'
: Do not print messages saying which checks are being made. To suppress all normal output, redirect it to '/dev/null' (any error messages will still be shown).

'--srcdir=DIR'
: Look for the package's source code in directory DIR. Usually 'configure' can determine that directory automatically.

'--version'
: Print the version of Autoconf used to generate the 'configure' script, and exit.

'configure' also accepts some other, not widely useful, options.

4. NEWS

Release 3.0.1 (29 August 2005)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3.0.1 fixes a bunch of bugs reported in 3.0.0. There is no new
functionality. Some of the fixed bugs are critical, so if you
use/distribute 3.0.0, an upgrade to 3.0.1 is recommended. The fixed
bugs are:

(note: "n-i-bz" means "not in bugzilla" -- this bug does not have
a bugzilla entry).

109313 (== 110505) x86 cmpxchg8b
n-i-bz x86: track but ignore changes to %eflags.AC (alignment check)
110102 dis_op2_E_G(amd64)
110202 x86 sys_waitpid(#286)
110203 clock_getres(,0)
110208 execve fail wrong retval
110274 SSE1 now mandatory for x86
110388 amd64 0xDD 0xD1
110464 amd64 0xDC 0x1D FCOMP
110478 amd64 0xF 0xD PREFETCH
n-i-bz XML <unique> printing wrong
n-i-bz Dirk r4359 (amd64 syscalls from trunk)
110591 amd64 and x86: rdtsc not implemented properly
n-i-bz Nick r4384 (stub implementations of Addrcheck and Helgrind)
110652 AMD64 valgrind crashes on cwtd instruction
110653 AMD64 valgrind crashes on sarb $0x4,foo(%rip) instruction
110656 PATH=/usr/bin::/bin valgrind foobar stats ./fooba
110657 Small test fixes
110671 vex x86->IR: unhandled instruction bytes: 0xF3 0xC3 (rep ret)
n-i-bz Nick (Cachegrind should not assert when it encounters a client
request.)
110685 amd64->IR: unhandled instruction bytes: 0xE1 0x56 (loope Jb)
110830 configuring with --host fails to build 32 bit on 64 bit target
110875 Assertion when execve fails
n-i-bz Updates to Memcheck manual
n-i-bz Fixed broken malloc_usable_size()
110898 opteron instructions missing: btq btsq btrq bsfq
110954 x86->IR: unhandled instruction bytes: 0xE2 0xF6 (loop Jb)
n-i-bz Make suppressions work for "???" lines in stacktraces.
111006 bogus warnings from linuxthreads
111092 x86: dis_Grp2(Reg): unhandled case(x86)
111231 sctp_getladdrs() and sctp_getpaddrs() returns uninitialized
memory
111102 (comment #4) Fixed 64-bit unclean "silly arg" message
n-i-bz vex x86->IR: unhandled instruction bytes: 0x14 0x0
n-i-bz minor umount/fcntl wrapper fixes
111090 Internal Error running Massif
101204 noisy warning
111513 Illegal opcode for SSE instruction (x86 movups)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

111555 VEX/Makefile: CC is set to gcc
n-i-bz Fix XML bugs in FAQ

(3.0.1: 29 August 05,
vex/branches/VEX_3_0_BRANCH r1367,
valgrind/branches/VALGRIND_3_0_BRANCH r4574).

Release 3.0.0 (3 August 2005)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

3.0.0 is a major overhaul of Valgrind. The most significant user visible change is that Valgrind now supports architectures other than x86. The new architectures it supports are AMD64 and PPC32, and the infrastructure is present for other architectures to be added later.

AMD64 support works well, but has some shortcomings:

- It generally won't be as solid as the x86 version. For example, support for more obscure instructions and system calls may be missing. We will fix these as they arise.

- Address space may be limited; see the point about position-independent executables below.

- If Valgrind is built on an AMD64 machine, it will only run 64-bit executables. If you want to run 32-bit x86 executables under Valgrind on an AMD64, you will need to build Valgrind on an x86 machine and copy it to the AMD64 machine. And it probably won't work if you do something tricky like exec'ing a 32-bit program from a 64-bit program while using --trace-children=yes. We hope to improve this situation in the future.

The PPC32 support is very basic. It may not work reliably even for small programs, but it's a start. Many thanks to Paul Mackerras for his great work that enabled this support. We are working to make PPC32 usable as soon as possible.

Other user-visible changes:

- Valgrind is no longer built by default as a position-independent executable (PIE), as this caused too many problems.

  Without PIE enabled, AMD64 programs will only be able to access 2GB of address space. We will fix this eventually, but not for the moment.

  Use --enable-pie at configure-time to turn this on.

- Support for programs that use stack-switching has been improved. Use the --max-stackframe flag for simple cases, and the VALGRIND_STACK_REGISTER, VALGRIND_STACK_DEREGISTER and VALGRIND_STACK_CHANGE client requests for trickier cases.

- Support for programs that use self-modifying code has been improved,
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

in particular programs that put temporary code fragments on the stack. This helps for C programs compiled with GCC that use nested functions, and also Ada programs. This is controlled with the --smc-check flag, although the default setting should work in most cases.

- Output can now be printed in XML format. This should make it easier for tools such as GUI front-ends and automated error-processing schemes to use Valgrind output as input. The --xml flag controls this. As part of this change, ELF directory information is read from executables, so absolute source file paths are available if needed.

- Programs that allocate many heap blocks may run faster, due to improvements in certain data structures.

- Addrcheck is currently not working. We hope to get it working again soon. Helgrind is still not working, as was the case for the 2.4.0 release.

- The JITter has been completely rewritten, and is now in a separate library, called Vex. This enabled a lot of the user-visible changes, such as new architecture support. The new JIT unfortunately translates more slowly than the old one, so programs may take longer to start. We believe the code quality is produces is about the same, so once started, programs should run at about the same speed. Feedback about this would be useful.

  On the plus side, Vex and hence Memcheck tracks value flow properly through floating point and vector registers, something the 2.X line could not do. That means that Memcheck is much more likely to be usably accurate on vectorised code.

- There is a subtle change to the way exiting of threaded programs is handled. In 3.0, Valgrind’s final diagnostic output (leak check, etc) is not printed until the last thread exits. If the last thread to exit was not the original thread which started the program, any other process wait()-ing on this one to exit may conclude it has finished before the diagnostic output is printed. This may not be what you expect. 2.X had a different scheme which avoided this problem, but caused deadlocks under obscure circumstances, so we are trying something different for 3.0.

- Small changes in control log file naming which make it easier to use valgrind for debugging MPI-based programs. The relevant new flags are --log-file-exactly= and --log-file-qualifier=.

- As part of adding AMD64 support, DWARF2 CFI-based stack unwinding support was added. In principle this means Valgrind can produce meaningful backtraces on x86 code compiled with -fomit-frame-pointer providing you also compile your code with -fasynchronous-unwind-tables.

- The documentation build system has been completely redone. The documentation masters are now in XML format, and from that HTML, PostScript and PDF documentation is generated. As a result the manual is now available in book form. Note that the

documentation in the source tarballs is pre-built, so you don't need
any XML processing tools to build Valgrind from a tarball.

Changes that are not user-visible:

- The code has been massively overhauled in order to modularise it. As a result we hope it is easier to navigate and understand.

- Lots of code has been rewritten.

BUGS FIXED:

110046	sz == 4 assertion failed
109810	vex amd64->IR: unhandled instruction bytes: 0xA3 0x4C 0x70 0xD7
109802	Add a plausible_stack_size command-line parameter ?
109783	unhandled ioctl TIOCMGET (running hw detection tool discover)
109780	unhandled ioctl BLKSSZGET (running fdisk -l /dev/hda)
109718	vex x86->IR: unhandled instruction: ffreep
109429	AMD64 unhandled syscall: 127 (sigpending)
109401	false positive uninit in strchr from ld-linux.so.2
109385	"stabs" parse failure
109378	amd64: unhandled instruction REP NOP
109376	amd64: unhandled instruction LOOP Jb
109363	AMD64 unhandled instruction bytes
109362	AMD64 unhandled syscall: 24 (sched_yield)
109358	fork() won't work with valgrind-3.0 SVN
109332	amd64 unhandled instruction: ADC Ev, Gv
109314	Bogus memcheck report on amd64
108883	Crash; vg_memory.c:905 (vgPlain_init_shadow_range): Assertion 'vgPlain_defined_init_shadow_page()' failed.
108349	mincore syscall parameter checked incorrectly
108059	build infrastructure: small update
107524	epoll_ctl event parameter checked on EPOLL_CTL_DEL
107123	Vex dies with unhandled instructions: 0xD9 0x31 0xF 0xAE
106841	auxmap & openGL problems
106713	SDL_Init causes valgrind to exit
106352	setcontext and makecontext not handled correctly
106293	addresses beyond initial client stack allocation not checked in VALGRIND_DO_LEAK_CHECK
106283	PIE client programs are loaded at address 0
105831	Assertion 'vgPlain_defined_init_shadow_page()' failed.
105039	long run-times probably due to memory manager
104797	valgrind needs to be aware of BLKGETSIZE64
103594	unhandled instruction: FICOM
103320	Valgrind 2.4.0 fails to compile with gcc 3.4.3 and -O0
103168	potentially memory leak in coregrind/ume.c
102039	bad permissions for mapped region at address 0xB7C73680
101881	weird assertion problem
101543	Support fadvise64 syscalls
75247	x86_64/amd64 support (the biggest "bug" we have ever fixed)

(3.0RC1: 27 July 05, vex r1303, valgrind r4283).
(3.0.0: 3 August 05, vex r1313, valgrind r4316).

Stable release 2.4.0 (March 2005) -- CHANGES RELATIVE TO 2.2.0

2.4.0 brings many significant changes and bug fixes. The most significant user-visible change is that we no longer supply our own pthread implementation. Instead, Valgrind is finally capable of running the native thread library, either LinuxThreads or NPTL.

This means our libpthread has gone, along with the bugs associated with it. Valgrind now supports the kernel's threading syscalls, and lets you use your standard system libpthread. As a result:

- There are many fewer system dependencies and strange library-related bugs. There is a small performance improvement, and a large stability improvement.

- On the downside, Valgrind can no longer report misuses of the POSIX PThreads API. It also means that Helgrind currently does not work. We hope to fix these problems in a future release.

Note that running the native thread libraries does not mean Valgrind is able to provide genuine concurrent execution on SMPs. We still impose the restriction that only one thread is running at any given time.

There are many other significant changes too:

- Memcheck is (once again) the default tool.

- The default stack backtrace is now 12 call frames, rather than 4.

- Suppressions can have up to 25 call frame matches, rather than 4.

- Memcheck and Addrcheck use less memory. Under some circumstances, they no longer allocate shadow memory if there are large regions of memory with the same A/V states - such as an mmaped file.

- The memory-leak detector in Memcheck and Addrcheck has been improved. It now reports more types of memory leak, including leaked cycles. When reporting leaked memory, it can distinguish between directly leaked memory (memory with no references), and indirectly leaked memory (memory only referred to by other leaked memory).

- Memcheck's confusion over the effect of mprotect() has been fixed: previously mprotect could erroneously mark undefined data as defined.

- Signal handling is much improved and should be very close to what you get when running natively.

  One result of this is that Valgrind observes changes to sigcontexts passed to signal handlers. Such modifications will take effect when the signal returns. You will need to run with --single-step=yes to make this useful.

* Valgrind is built in Position Independent Executable (PIE) format if your toolchain supports it. This allows it to take advantage of all the available address space on systems with 4Gbyte user address spaces.

* Valgrind can now run itself (requires PIE support).

* Syscall arguments are now checked for validity. Previously all memory used by syscalls was checked, but now the actual values passed are also checked.

* Syscall wrappers are more robust against bad addresses being passed to syscalls: they will fail with EFAULT rather than killing Valgrind with SIGSEGV.

* Because clone() is directly supported, some non-pthread uses of it will work. Partial sharing (where some resources are shared, and some are not) is not supported.

* open() and readlink() on /proc/self/exe are supported.

BUGS FIXED:

88520	pipe+fork+dup2 kills the main program
88604	Valgrind Aborts when using $VALGRIND_OPTS and user progra...
88614	valgrind: vg_libpthread.c:2323 (read): Assertion ‘read_pt...
88703	Stabs parser fails to handle ";"
88886	ioctl wrappers for TIOCMBIS and TIOCMBIC
89032	valgrind pthread_cond_timedwait fails
89106	the ’impossible’ happened
89139	Missing sched_setaffinity & sched_getaffinity
89198	valgrind lacks support for SIOCSPGRP and SIOCGPGRP
89263	Missing ioctl translations for scsi-generic and CD playing
89440	tests/deadlock.c line endings
89481	‘impossible’ happened: EXEC FAILED
89663	valgrind 2.2.0 crash on Redhat 7.2
89792	Report pthread_mutex_lock() deadlocks instead of returnin...
90111	statvfs64 gives invalid error/warning
90128	crash+memory fault with stabs generated by gnat for a run...
90778	VALGRIND_CHECK_DEFINED() not as documented in memcheck.h
90834	cachegrind crashes at end of program without reporting re...
91028	valgrind: vg_memory.c:229 (vgPlain_unmap_range): Assertio...
91162	valgrind crash while debugging drivel 1.2.1
91199	Unimplemented function
91325	Signal routing does not propagate the siginfo structure
91599	Assertion ‘cv == ((void *)0)’
91604	rw_lookup clears orig and sends the NULL value to rw_new
91821	Small problems building valgrind with $top_builddir ne $t...
91844	signal 11 (SIGSEGV) at get_tcb (libpthread.c:86) in corec...
92264	UNIMPLEMENTED FUNCTION: pthread_condattr_setpshared
92331	per-target flags necessitate AM_PROG_CC_C_O

92420 valgrind doesn’t compile with linux 2.6.8.1/9
92513 Valgrind 2.2.0 generates some warning messages
92528 vg_symtab2.c:170 (addLoc): Assertion ‘loc->size > 0’ failed.
93096 unhandled ioctl 0x4B3A and 0x5601
93117 Tool and core interface versions do not match
93128 Can’t run valgrind --tool=memcheck because of unimplement...
93174 Valgrind can crash if passed bad args to certain syscalls
93309 Stack frame in new thread is badly aligned
93328 Wrong types used with sys_sigprocmask()
93763 /usr/include/asm/msr.h is missing
93776 valgrind: vg_memory.c:508 (vgPlain_find_map_space): Asser...
93810 fcntl() argument checking a bit too strict
94378 Assertion ‘tst->sigqueue_head != tst->sigqueue_tail’ failed.
94429 valgrind 2.2.0 segfault with mmap64 in glibc 2.3.3
94645 Impossible happened: PINSRW mem
94953 valgrind: the ‘impossible’ happened: SIGSEGV
95667 Valgrind does not work with any KDE app
96243 Assertion ’res==0’ failed
96252 stage2 loader of valgrind fails to allocate memory
96520 All programs crashing at _dl_start (in /lib/ld-2.3.3.so) ...
96660 ioctl CDROMREADTOCENTRY causes bogus warnings
96747 After looping in a segfault handler, the impossible happens
96923 Zero sized arrays crash valgrind trace back with SIGFPE
96948 valgrind stops with assertion failure regarding mmap2
96966 valgrind fails when application opens more than 16 sockets
97398 valgrind: vg_libpthread.c:2667 Assertion failed
97407 valgrind: vg_mylibc.c:1226 (vgPlain_safe_fd): Assertion ‘...
97427 "Warning: invalid file descriptor -1 in syscall close()" ...
97785 missing backtrace
97792 build in obj dir fails - autoconf / makefile cleanup
97880 pthread_mutex_lock fails from shared library (special ker...
97975 program aborts without ang VG messages
98129 Failed when open and close file 230000 times using stdio
98175 Crashes when using valgrind-2.2.0 with a program using al...
98288 Massif broken
98303 UNIMPLEMENTED FUNCTION pthread_condattr_setpshared
98630 failed--compilation missing warnings.pm, fails to make he...
98756 Cannot valgrind signal-heavy kdrive X server
98966 valgrinding the JVM fails with a sanity check assertion
99035 Valgrind crashes while profiling
99142 loops with message "Signal 11 being dropped from thread 0...
99195 threaded apps crash on thread start (using QThread::start...
99348 Assertion ‘vgPlain_lseek(core_fd, 0, 1) == phdrs[i].p_off...
99568 False negative due to mishandling of mprotect
99738 valgrind memcheck crashes on program that uses sigitimer
99923 0-sized allocations are reported as leaks
99949 program seg faults after exit()
100036 "newSuperblock’s request for 1048576 bytes failed"
100116 valgrind: (pthread_cond_init): Assertion ‘sizeof(* cond) ...
100486 memcheck reports "valgrind: the ‘impossible’ happened: V...
100833 second call to "mremap" fails with EINVAL
101156 (vgPlain_find_map_space): Assertion ‘(addr & ((1 << 12)-1...
101173 Assertion ‘recDepth >= 0 && recDepth < 500’ failed
101291 creating threads in a forked process fails

101313 valgrind causes different behavior when resizing a window...
101423 segfault for c++ array of floats
101562 valgrind massif dies on SIGINT even with signal handler r...

Stable release 2.2.0 (31 August 2004) -- CHANGES RELATIVE TO 2.0.0
~~~~~~

2.2.0 brings nine months worth of improvements and bug fixes. We believe it to be a worthy successor to 2.0.0. There are literally hundreds of bug fixes and minor improvements. There are also some fairly major user-visible changes:

* A complete overhaul of handling of system calls and signals, and their interaction with threads. In general, the accuracy of the system call, thread and signal simulations is much improved:

  - Blocking system calls behave exactly as they do when running natively (not on valgrind). That is, if a syscall blocks only the calling thread when running natively, than it behaves the same on valgrind. No more mysterious hangs because V doesn't know that some syscall or other, should block only the calling thread.

  - Interrupted syscalls should now give more faithful results.

  - Signal contexts in signal handlers are supported.

* Improvements to NPTL support to the extent that V now works properly on NPTL-only setups.

* Greater isolation between Valgrind and the program being run, so the program is less likely to inadvertently kill Valgrind by doing wild writes.

* Massif: a new space profiling tool. Try it! It's cool, and it'll tell you in detail where and when your C/C++ code is allocating heap. Draws pretty .ps pictures of memory use against time. A potentially powerful tool for making sense of your program's space use.

* File descriptor leakage checks. When enabled, Valgrind will print out a list of open file descriptors on exit.

* Improved SSE2/SSE3 support.

* Time-stamped output; use --time-stamp=yes

Stable release 2.2.0 (31 August 2004) -- CHANGES RELATIVE TO 2.1.2
~~~~~~

2.2.0 is not much different from 2.1.2, released seven weeks ago. A number of bugs have been fixed, most notably #85658, which gave problems for quite a few people. There have been many internal cleanups, but those are not user visible.

The following bugs have been fixed since 2.1.2:

85658 Assert in coregrind/vg_libpthread.c:2326 (open64) != (void*)0 failed
This bug was reported multiple times, and so the following duplicates of it are also fixed: 87620, 85796, 85935, 86065, 86919, 86988, 87917, 88156

80716 Semaphore mapping bug caused by unmap (sem_destroy)
(Was fixed prior to 2.1.2)

86987 semctl and shmctl syscalls family is not handled properly

86696 valgrind 2.1.2 + RH AS2.1 + librt

86730 valgrind locks up at end of run with assertion failure in __pthread_unwind

86641 memcheck doesn't work with Mesa OpenGL/ATI on Suse 9.1
(also fixes 74298, a duplicate of this)

85947 MMX/SSE unhandled instruction 'sfence'

84978 Wrong error "Conditional jump or move depends on uninitialised value" resulting from "sbbl %reg, %reg"

86254 ssort() fails when signed int return type from comparison is too small to handle result of unsigned int subtraction

87089 memalign(4, xxx) makes valgrind assert

86407 Add support for low-level parallel port driver ioctls.

70587 Add timestamps to Valgrind output? (wishlist)

84937 vg_libpthread.c:2505 (se_remap): Assertion ‘res == 0’
(fixed prior to 2.1.2)

86317 cannot load libSDL-1.2.so.0 using valgrind

86989 memcpy from mac_replace_strmem.c complains about uninitialized pointers passed when length to copy is zero

85811 gnu pascal symbol causes segmentation fault; ok in 2.0.0

79138 writing to sbrk()'d memory causes segfault

77369 sched deadlock while signal received during pthread_join and the joined thread exited

88115 In signal handler for SIGFPE, siginfo->si_addr is wrong under Valgrind

78765 Massif crashes on app exit if FP exceptions are enabled

Additionally there are the following changes, which are not connected to any bug report numbers, AFAICS:

* Fix scary bug causing mis-identification of SSE stores vs
  loads and so causing memcheck to sometimes give nonsense results
  on SSE code.

* Add support for the POSIX message queue system calls.

* Fix to allow 32-bit Valgrind to run on AMD64 boxes. Note: this does
  NOT allow Valgrind to work with 64-bit executables - only with 32-bit
  executables on an AMD64 box.

* At configure time, only check whether linux/mii.h can be processed
  so that we don't generate ugly warnings by trying to compile it.

* Add support for POSIX clocks and timers.

Developer (cvs head) release 2.1.2 (18 July 2004)
~~~~~~

2.1.2 contains four months worth of bug fixes and refinements. Although officially a developer release, we believe it to be stable enough for widespread day-to-day use. 2.1.2 is pretty good, so try it first, although there is a chance it won't work. If so then try 2.0.0 and tell us what went wrong." 2.1.2 fixes a lot of problems present in 2.0.0 and is generally a much better product.

Relative to 2.1.1, a large number of minor problems with 2.1.1 have been fixed, and so if you use 2.1.1 you should try 2.1.2. Users of the last stable release, 2.0.0, might also want to try this release.

The following bugs, and probably many more, have been fixed. These are listed at http://bugs.kde.org. Reporting a bug for valgrind in the http://bugs.kde.org is much more likely to get you a fix than mailing developers directly, so please continue to keep sending bugs there.

76869 Crashes when running any tool under Fedora Core 2 test1
This fixes the problem with returning from a signal handler
when VDSOs are turned off in FC2.

69508 java 1.4.2 client fails with erroneous "stack size too small".
This fix makes more of the pthread stack attribute related
functions work properly. Java still doesn't work though.

71906 malloc alignment should be 8, not 4
All memory returned by malloc/new etc is now at least
8-byte aligned.

81970 vg_alloc_ThreadState: no free slots available
(closed because the workaround is simple: increase
~~~~~~

VG_N_THREADS, rebuild and try again.)

78514 Conditional jump or move depends on uninitialized value(s)
(a slight mishanding of FP code in memcheck)

77952 pThread Support (crash) (due to initialisation-ordering probs)
(also 85118)

80942 Addrcheck wasn’t doing overlap checking as it should.
78048 return NULL on malloc/new etc failure, instead of asserting
73655 operator new() override in user .so files often doesn’t get picked up
83060 Valgrind does not handle native kernel AIO
69872 Create proper coredumps after fatal signals
82026 failure with new glibc versions: __libc_* functions are not exported
70344 UNIMPLEMENTED FUNCTION: tcdrain
81297 Cancellation of pthread_cond_wait does not require mutex
82872 Using debug info from additional packages (wishlist)
83025 Support for ioctls FIGETBSZ and FIBMAP
83340 Support for ioctl HDIO_GET_IDENTITY
79714 Support for the semtimedop system call.
77022 Support for ioctls FBIOGET_VSCREENINFO and FBIOGET_FSCREENINFO
82098 hp2ps ansification (wishlist)
83573 Valgrind SIGSEGV on execve
82999 show which cmdline option was erroneous (wishlist)
83040 make valgrind VPATH and distcheck-clean (wishlist)
83998 Assertion ‘newfd > vgPlain_max_fd’ failed (see below)
82722 Unchecked mmap in as_pad leads to mysterious failures later
78958 memcheck seg faults while running Mozilla
85416 Arguments with colon (e.g. --logsocket) ignored

Additionally there are the following changes, which are not connected to any bug report numbers, AFAICS:

* Rearranged address space layout relative to 2.1.1, so that Valgrind/tools will run out of memory later than currently in many circumstances. This is good news esp. for Calltree. It should be possible for client programs to allocate over 800MB of memory when using memcheck now.

* Improved checking when laying out memory. Should hopefully avoid the random segmentation faults that 2.1.1 sometimes caused.

* Support for Fedora Core 2 and SuSE 9.1. Improvements to NPTL support to the extent that V now works properly on NPTL-only setups.

* Renamed the following options:
  --logfile-fd --> --log-fd
  --logfile --> --log-file
  --logsocket --> --log-socket
  to be consistent with each other and other options (esp. --input-fd).

* Add support for SIOCGMIIPHY, SIOCGMIIREG and SIOCSMIIREG ioctls and improve the checking of other interface related ioctls.

* Fix building with gcc-3.4.1.

* Remove limit on number of semaphores supported.

* Add support for syscalls: set_tid_address (258), acct (51).

* Support instruction "repne movs" -- not official but seems to occur.

* Implement an emulated soft limit for file descriptors in addition to
  the current reserved area, which effectively acts as a hard limit. The
  setrlimit system call now simply updates the emulated limits as best
  as possible - the hard limit is not allowed to move at all and just
  returns EPERM if you try and change it. This should stop reductions
  in the soft limit causing assertions when valgrind tries to allocate
  descriptors from the reserved area.
  (This actually came from bug #83998).

* Major overhaul of Cachegrind implementation. First user-visible change
  is that cachegrind.out files are now typically 90% smaller than they
  used to be; code annotation times are correspondingly much smaller.
  Second user-visible change is that hit/miss counts for code that is
  unloaded at run-time is no longer dumped into a single "discard" pile,
  but accurately preserved.

* Client requests for telling valgrind about memory pools.

Developer (cvs head) release 2.1.1 (12 March 2004)
~~~~~~
2.1.1 contains some internal structural changes needed for V's
long-term future. These don't affect end-users. Most notable
user-visible changes are:

* Greater isolation between Valgrind and the program being run, so
  the program is less likely to inadvertently kill Valgrind by
  doing wild writes.

* Massif: a new space profiling tool. Try it! It's cool, and it'll
  tell you in detail where and when your C/C++ code is allocating heap.
  Draws pretty .ps pictures of memory use against time. A potentially
  powerful tool for making sense of your program's space use.

* Fixes for many bugs, including support for more SSE2/SSE3 instructions,
  various signal/syscall things, and various problems with debug
  info readers.

* Support for glibc-2.3.3 based systems.

We are now doing automatic overnight build-and-test runs on a variety
of distros. As a result, we believe 2.1.1 builds and runs on:
Red Hat 7.2, 7.3, 8.0, 9, Fedora Core 1, SuSE 8.2, SuSE 9.
~~~~~~

The following bugs, and probably many more, have been fixed. These are listed at http://bugs.kde.org. Reporting a bug for valgrind in the http://bugs.kde.org is much more likely to get you a fix than mailing developers directly, so please continue to keep sending bugs there.

69616	glibc 2.3.2 w/NPTL is massively different than what valgrind expects
69856	I don't know how to instrument MMXish stuff (Helgrind)
73892	valgrind segfaults starting with Objective-C debug info (fix for S-type stabs)
73145	Valgrind complains too much about close(<reserved fd>)
73902	Shadow memory allocation seems to fail on RedHat 8.0
68633	VG_N_SEMAPHORES too low (V itself was leaking semaphores)
75099	impossible to trace multiprocess programs
76839	the ‘impossible’ happened: disInstr: INT but not 0x80 !
76762	vg_to_ucode.c:3748 (dis_push_segreg): Assertion ‘sz == 4’ failed.
76747	cannot include valgrind.h in c++ program
76223	parsing B(3,10) gave NULL type => impossible happens
75604	shmdt handling problem
76416	Problems with gcc 3.4 snap 20040225
75614	using -gstabs when building your programs the ‘impossible’ happened
75787	Patch for some CDROM ioctls CDORM_GET_MCN, CDROM_SEND_PACKET,
75294	gcc 3.4 snapshot's libstdc++ have unsupported instructions. (REP RET)
73326	vg_symtab2.c:272 (addScopeRange): Assertion ‘range->size > 0’ failed.
72596	not recognizing __libc_malloc
69489	Would like to attach ddd to running program
72781	Cachegrind crashes with kde programs
73055	Illegal operand at DXTCV11CompressBlockSSE2 (more SSE opcodes)
73026	Descriptor leak check reports port numbers wrongly
71705	README_MISSING_SYSCALL_OR_IOCTL out of date
72643	Improve support for SSE/SSE2 instructions
72484	valgrind leaves it's own signal mask in place when execing
72650	Signal Handling always seems to restart system calls
72006	The mmap system call turns all errors in ENOMEM
71781	gdb attach is pretty useless
71180	unhandled instruction bytes: 0xF 0xAE 0x85 0xE8
69886	writes to zero page cause valgrind to assert on exit
71791	crash when valgrinding gimp 1.3 (stabs reader problem)
69783	unhandled syscall: 218
69782	unhandled instruction bytes: 0x66 0xF 0x2B 0x80
70385	valgrind fails if the soft file descriptor limit is less than about 828
69529	"rep; nop" should do a yield
70827	programs with lots of shared libraries report "mmap failed" for some of them when reading symbols
71028	glibc's strnlen is optimised enough to confuse valgrind

Unstable (cvs head) release 2.1.0 (15 December 2003)

~~~~~~
~~~~~~

For whatever it's worth, 2.1.0 actually seems pretty darn stable to me (Julian). It looks eminently usable, and given that it fixes some significant bugs, may well be worth using on a day-to-day basis.
2.1.0 is known to build and pass regression tests on: SuSE 9, SuSE 8.2, RedHat 8.

2.1.0 most notably includes Jeremy Fitzhardinge's complete overhaul of handling of system calls and signals, and their interaction with threads. In general, the accuracy of the system call, thread and signal simulations is much improved. Specifically:

- Blocking system calls behave exactly as they do when running natively (not on valgrind). That is, if a syscall blocks only the calling thread when running natively, than it behaves the same on valgrind. No more mysterious hangs because V doesn't know that some syscall or other, should block only the calling thread.

- Interrupted syscalls should now give more faithful results.

- Finally, signal contexts in signal handlers are supported. As a result, konqueror on SuSE 9 no longer segfaults when notified of file changes in directories it is watching.

Other changes:

- Robert Walsh's file descriptor leakage checks. When enabled, Valgrind will print out a list of open file descriptors on exit. Along with each file descriptor, Valgrind prints out a stack backtrace of where the file was opened and any details relating to the file descriptor such as the file name or socket details.
  To use, give: --track-fds=yes

- Implemented a few more SSE/SSE2 instructions.

- Less crud on the stack when you do 'where' inside a GDB attach.

- Fixed the following bugs:
  68360: Valgrind does not compile against 2.6.0-testX kernels
  68525: CVS head doesn't compile on C90 compilers
  68566: pkgconfig support (wishlist)
  68588: Assertion 'sz == 4' failed in vg_to_ucode.c (disInstr)
  69140: valgrind not able to explicitly specify a path to a binary.
  69432: helgrind asserts encountering a MutexErr when there are
  EraserErr suppressions

- Increase the max size of the translation cache from 200k average bbs to 300k average bbs. Programs on the size of OOo (680m17) are thrashing the cache at the smaller size, creating large numbers of retranslations and wasting significant time as a result.

Stable release 2.0.0 (5 Nov 2003)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

2.0.0 improves SSE/SSE2 support, fixes some minor bugs, and improves support for SuSE 9 and the Red Hat "Severn" beta.

- Further improvements to SSE/SSE2 support. The entire test suite of the GNU Scientific Library (gsl-1.4) compiled with Intel Icc 7.1 20030307Z '-g -O -xW' now works. I think this gives pretty good coverage of SSE/SSE2 floating point instructions, or at least the subset emitted by Icc.

- Also added support for the following instructions:
  MOVNTDQ UCOMISD UNPCKLPS UNPCKHPS SQRTSS
  PUSH/POP %{FS,GS}, and PUSH %CS (Nb: there is no POP %CS).

- CFI support for GDB version 6. Needed to enable newer GDBs to figure out where they are when using --gdb-attach=yes.

- Fix this:
  mc_translate.c:1091 (memcheck_instrument): Assertion
  'u_in->size == 4 || u_in->size == 16' failed.

- Return an error rather than panicing when given a bad socketcall.

- Fix checking of syscall rt_sigtimedwait().

- Implement __NR_clock_gettime (syscall 265). Needed on Red Hat Severn.

- Fixed bug in overlap check in strncpy() -- it was assuming the src was 'n' bytes long, when it could be shorter, which could cause false positives.

- Support use of select() for very large numbers of file descriptors.

- Don't fail silently if the executable is statically linked, or is setuid/setgid. Print an error message instead.

- Support for old DWARF-1 format line number info.

Snapshot 20031012 (12 October 2003)

Three months worth of bug fixes, roughly. Most significant single change is improved SSE/SSE2 support, mostly thanks to Dirk Mueller.

20031012 builds on Red Hat Fedora ("Severn") but doesn't really work (curiosly, mozilla runs OK, but a modest "ls -l" bombs). I hope to get a working version out soon. It may or may not work ok on the forthcoming SuSE 9; I hear positive noises about it but haven't been able to verify this myself (not until I get hold of a copy of 9).

A detailed list of changes, in no particular order:

- Describe --gen-suppressions in the FAQ.

- Syscall __NR_waitpid supported.

- Minor MMX bug fix.

- -v prints program's argv[] at startup.

- More glibc-2.3 suppressions.

- Suppressions for stack underrun bug(s) in the c++ support library distributed with Intel Icc 7.0.

- Fix problems reading /proc/self/maps.

- Fix a couple of messages that should have been suppressed by -q, but weren't.

- Make Addrcheck understand "Overlap" suppressions.

- At startup, check if program is statically linked and bail out if so.

- Cachegrind: Auto-detect Intel Pentium-M, also VIA Nehemiah

- Memcheck/addrcheck: minor speed optimisations

- Handle syscall __NR_brk more correctly than before.

- Fixed incorrect allocate/free mismatch errors when using
  operator new(unsigned, std::nothrow_t const&)
  operator new[](unsigned, std::nothrow_t const&)

- Support POSIX pthread spinlocks.

- Fixups for clean compilation with gcc-3.3.1.

- Implemented more opcodes:
  - push %es
  - push %ds
  - pop %es
  - pop %ds
  - movntq
  - sfence
  - pshufw
  - pavgb
  - ucomiss
  - enter
  - mov imm32, %esp
  - all "in" and "out" opcodes
  - inc/dec %esp
  - A whole bunch of SSE/SSE2 instructions

- Memcheck: don't bomb on SSE/SSE2 code.

Snapshot 20030725 (25 July 2003)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Fixes some minor problems in 20030716.

- Fix bugs in overlap checking for strcpy/memcpy etc.

- Do overlap checking with Addrcheck as well as Memcheck.

- Fix this:
  Memcheck: the 'impossible' happened:
  get_error_name: unexpected type

- Install headers needed to compile new skins.

- Remove leading spaces and colon in the LD_LIBRARY_PATH / LD_PRELOAD passed to non-traced children.

- Fix file descriptor leak in valgrind-listener.

- Fix longstanding bug in which the allocation point of a block resized by realloc was not correctly set. This may have caused confusing error messages.

Snapshot 20030716 (16 July 2003)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

20030716 is a snapshot of our current CVS head (development) branch. This is the branch which will become valgrind-2.0. It contains significant enhancements over the 1.9.X branch.

Despite this being a snapshot of the CVS head, it is believed to be quite stable -- at least as stable as 1.9.6 or 1.0.4, if not more so -- and therefore suitable for widespread use. Please let us know asap if it causes problems for you.

Two reasons for releasing a snapshot now are:

- It's been a while since 1.9.6, and this snapshot fixes various problems that 1.9.6 has with threaded programs on glibc-2.3.X based systems.

- So as to make available improvements in the 2.0 line.

Major changes in 20030716, as compared to 1.9.6:

- More fixes to threading support on glibc-2.3.1 and 2.3.2-based systems (SuSE 8.2, Red Hat 9). If you have had problems with inconsistent/illogical behaviour of errno, h_errno or the DNS resolver functions in threaded programs, 20030716 should improve matters. This snapshot seems stable enough to run OpenOffice.org 1.1rc on Red Hat 7.3, SuSE 8.2 and Red Hat 9, and that's a big threaded app if ever I saw one.

- Automatic generation of suppression records; you no longer need to write them by hand. Use --gen-suppressions=yes.

- strcpy/memcpy/etc check their arguments for overlaps, when running with the Memcheck or Addrcheck skins.

- malloc_usable_size() is now supported.

- new client requests:
    - VALGRIND_COUNT_ERRORS, VALGRIND_COUNT_LEAKS: useful with regression testing
    - VALGRIND_NON_SIMD_CALL[0123]: for running arbitrary functions on real CPU (use with caution!)

- The GDB attach mechanism is more flexible. Allow the GDB to be run to be specified by --gdb-path=/path/to/gdb, and specify which file descriptor V will read its input from with --input-fd=<number>.

- Cachegrind gives more accurate results (wasn't tracking instructions in malloc() and friends previously, is now).

- Complete support for the MMX instruction set.

- Partial support for the SSE and SSE2 instruction sets. Work for this is ongoing. About half the SSE/SSE2 instructions are done, so some SSE based programs may work. Currently you need to specify --skin=addrcheck. Basically not suitable for real use yet.

- Significant speedups (10%-20%) for standard memory checking.

- Fix assertion failure in pthread_once().

- Fix this:

```
    valgrind: vg_intercept.c:598 (vgAllRoadsLeadToRome_select):
              Assertion `ms_end >= ms_now' failed.
```

- Implement pthread_mutexattr_setpshared.

- Understand Pentium 4 branch hints. Also implemented a couple more obscure x86 instructions.

- Lots of other minor bug fixes.

- We have a decent regression test system, for the first time. This doesn't help you directly, but it does make it a lot easier for us to track the quality of the system, especially across multiple linux distributions.

  You can run the regression tests with 'make regtest' after 'make install' completes. On SuSE 8.2 and Red Hat 9 I get this:

```
     == 84 tests, 0 stderr failures, 0 stdout failures ==
```

On Red Hat 8, I get this:

```
     == 84 tests, 2 stderr failures, 1 stdout failure ==
     corecheck/tests/res_search               (stdout)
     memcheck/tests/sigaltstack               (stderr)
```

sigaltstack is probably harmless. res_search doesn't work
on R H 8 even running natively, so I'm not too worried.

On Red Hat 7.3, a glibc-2.2.5 system, I get these harmless failures:

```
     == 84 tests, 2 stderr failures, 1 stdout failure ==
     corecheck/tests/pth_atfork1              (stdout)
     corecheck/tests/pth_atfork1              (stderr)
     memcheck/tests/sigaltstack               (stderr)
```

You need to run on a PII system, at least, since some tests contain P6-specific instructions, and the test machine needs access to the internet so that corecheck/tests/res_search (a test that the DNS resolver works) can function.

As ever, thanks for the vast amount of feedback :) and bug reports :(We may not answer all messages, but we do at least look at all of them, and tend to fix the most frequently reported bugs.

Version 1.9.6 (7 May 2003 or thereabouts)

~~~~~~

Major changes in 1.9.6:

- Improved threading support for glibc >= 2.3.2 (SuSE 8.2, RedHat 9, to name but two ...) It turned out that 1.9.5 had problems with threading support on glibc >= 2.3.2, usually manifested by threaded programs deadlocking in system calls, or running unbelievably slowly. Hopefully these are fixed now. 1.9.6 is the first valgrind which gives reasonable support for glibc-2.3.2. Also fixed a 2.3.2 problem with pthread_atfork().

- Majorly expanded FAQ.txt. We've added workarounds for all common problems for which a workaround is known.

Minor changes in 1.9.6:

- Fix identification of the main thread's stack. Incorrect identification of it was causing some on-stack addresses to not get identified as such. This only affected the usefulness of some error messages; the correctness of the checks made is unchanged.

- Support for kernels >= 2.5.68.
~~~~~~

- Dummy implementations of __libc_current_sigrtmin, __libc_current_sigrtmax and __libc_allocate_rtsig, hopefully good enough to keep alive programs which previously died for lack of them.

- Fix bug in the VALGRIND_DISCARD_TRANSLATIONS client request.

- Fix bug in the DWARF2 debug line info loader, when instructions following each other have source lines far from each other (e.g. with inlined functions).

- Debug info reading: read symbols from both "symtab" and "dynsym" sections, rather than merely from the one that comes last in the file.

- New syscall support: prctl(), creat(), lookup_dcookie().

- When checking calls to accept(), recvfrom(), getsocketopt(), don’t complain if buffer values are NULL.

- Try and avoid assertion failures in mash_LD_PRELOAD_and_LD_LIBRARY_PATH.

- Minor bug fixes in cg_annotate.

Version 1.9.5 (7 April 2003)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

It occurs to me that it would be helpful for valgrind users to record in the source distribution the changes in each release. So I now attempt to mend my errant ways :-) Changes in this and future releases will be documented in the NEWS file in the source distribution.

Major changes in 1.9.5:

- (Critical bug fix): Fix a bug in the FPU simulation. This was causing some floating point conditional tests not to work right. Several people reported this. If you had floating point code which didn’t work right on 1.9.1 to 1.9.4, it’s worth trying 1.9.5.

- Partial support for Red Hat 9. RH9 uses the new Native Posix Threads Library (NPTL), instead of the older LinuxThreads. This potentially causes problems with V which will take some time to correct. In the meantime we have partially worked around this, and so 1.9.5 works on RH9. Threaded programs still work, but they may deadlock, because some system calls (accept, read, write, etc) which should be nonblocking, in fact do block. This is a known bug which we are looking into.

  If you can, your best bet (unfortunately) is to avoid using 1.9.5 on a Red Hat 9 system, or on any NPTL-based distribution. If your glibc is 2.3.1 or earlier, you’re almost certainly OK.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Minor changes in 1.9.5:

- Added some #errors to valgrind.h to ensure people don't include it accidentally in their sources. This is a change from 1.0.X which was never properly documented. The right thing to include is now memcheck.h. Some people reported problems and strange behaviour when (incorrectly) including valgrind.h in code with 1.9.1 -- 1.9.4. This is no longer possible.

- Add some __extension__ bits and pieces so that gcc configured for valgrind-checking compiles even with -Werror. If you don't understand this, ignore it. Of interest to gcc developers only.

- Removed a pointless check which caused problems interworking with Clearcase. V would complain about shared objects whose names did not end ".so", and refuse to run. This is now fixed. In fact it was fixed in 1.9.4 but not documented.

- Fixed a bug causing an assertion failure of "waiters == 1" somewhere in vg_scheduler.c, when running large threaded apps, notably MySQL.

- Add support for the munlock system call (124).

Some comments about future releases:

1.9.5 is, we hope, the most stable Valgrind so far. It pretty much supersedes the 1.0.X branch. If you are a valgrind packager, please consider making 1.9.5 available to your users. You can regard the 1.0.X branch as obsolete: 1.9.5 is stable and vastly superior. There are no plans at all for further releases of the 1.0.X branch.

If you want a leading-edge valgrind, consider building the cvs head (from SourceForge), or getting a snapshot of it. Current cool stuff going in includes MMX support (done); SSE/SSE2 support (in progress), a significant (10-20%) performance improvement (done), and the usual large collection of minor changes. Hopefully we will be able to improve our NPTL support, but no promises.

5. 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.

Valgrind works on most, reasonably recent Linux setups. If you have
problems, consult FAQ.txt to see if there are workarounds.

Executive Summary
~~~~~~~~~~~~~~~~~
Valgrind is an award-winning suite of tools for debugging and profiling
Linux programs. With the tools that come with Valgrind, you can
automatically detect many memory management and threading bugs, avoiding
hours of frustrating bug-hunting, making your programs more stable. You can
also perform detailed profiling, to speed up and reduce memory use of your
programs.

The Valgrind distribution currently includes five tools: two memory error
detectors, a thread error detector, a cache profiler and a heap profiler.

To give you an idea of what Valgrind tools do, when a program is run
under the supervision of the first memory error detector tool, all reads
and writes of memory are checked, and calls to malloc/new/free/delete
are intercepted. As a result, it can detect problems such as:

   Use of uninitialised memory
   Reading/writing memory after it has been free'd
   Reading/writing off the end of malloc'd blocks
   Reading/writing inappropriate areas on the stack
   Memory leaks -- where pointers to malloc'd blocks are lost forever
   Passing of uninitialised and/or unaddressible memory to system calls
   Mismatched use of malloc/new/new [] vs free/delete/delete []
   Overlaps of arguments to strcpy() and related functions
   Some abuses of the POSIX pthread API

Problems like these can be difficult to find by other means, often
lying undetected for long periods, then causing occasional,
difficult-to-diagnose crashes. When one of these errors occurs, you can
attach GDB to your program, so you can poke around and see what's going
on.

Valgrind is closely tied to details of the CPU, operating system and
to a less 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 and PPC32/Linux.

Valgrind is licensed under the GNU General Public License, version 2. Read the file COPYING in the source distribution for details.

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 Subversion repository :

0. Check out the code from SVN, following the instructions at http://valgrind.org/devel/cvs_svn.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 standard options are documented in the INSTALL file. The only interesting one is the usual --prefix=/where/you/want/it/installed.

5. Do "make".

6. Do "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 www.valgrind.org).

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.

Julian Seward (jseward@acm.org)
Nick Nethercote (njn@valgrind.org)
Jeremy Fitzhardinge (jeremy@goop.org)

6. 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/vg_syscalls.c.

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 fulfills 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 writeable. 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()

Removing the debug printing clutter, it looks like this:

```
   PRE(time)
   {
      /* time_t time(time_t *t); */
      PRINT("time ( %p )",arg1);
      if (arg1 != (UWord)NULL) {
         PRE_MEM_WRITE( "time", arg1, sizeof(time_t) );
      }
   }

   POST(time)
   {
      if (arg1 != (UWord)NULL) {
         POST_MEM_WRITE( arg1, sizeof(vki_time_t) );
      }
   }
```

The first thing we do happens before the syscall occurs, in the PRE() function: 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 != (UWord)NULL) {
         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 != (UInt)NULL) {
         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 coregrind/$(VG_PLATFORM)/vki_unistd.h.
~~~~~~

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 coregrind/vg_syscalls.c. 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 sys_info[] array; use SYSBA if it requires a PRE() and POST() function, and SYSB_ if it only requires a PRE() function. The 2nd arg of these macros indicate if the syscall could possibly block.

   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.

7. README_DEVELOPERS

Building and not installing it
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

To run Valgrind without having to install it, run coregrind/valgrind
with the VALGRINDLIB environment variable set, where <dir> is the root
of the source tree (and must be an absolute path). Eg:

```
VALGRINDLIB=~/grind/head4/.in_place ~/grind/head4/coregrind/valgrind
```

This allows you to compile and run with "make" instead of "make install",
saving you time.

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.

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
```

Debugging Valgrind with GDB
~~~~~~~~~~~~~~~~~~~~~~~~~~~

To debug stage 1 just run it under GDB in the normal way.

To debug Valgrind proper (stage 2) with GDB, start Valgrind like this:

```
valgrind --tool=none --wait-for-gdb=yes <prog>
```

Then start gdb like this in another terminal:

```
gdb /usr/lib/valgrind/stage2 <pid>
```

Where <pid> is the pid valgrind printed. Then set whatever breakpoints
you want and do this in gdb:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
jump  *$eip
```

8. 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.

-- (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 stage2 or libpthread.so. Valgrind will still work if you do, but it 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(). If stage2 was stripped the message would look like this:

```
Mismatched free() / delete / delete []
   at 0x40043249: (inside stage2)
   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 stage2)
   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.

-- Please test the final installation works by running it on
something huge. I suggest checking that it can start and
exit successfully both Mozilla-1.0 and OpenOffice.org 1.0.
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 --workaround-gcc296-bugs=yes mozilla

valgrind -v --trace-children=yes --workaround-gcc296-bugs=yes soffice
```

If you find any more hints/tips for packaging, please report
it as a bugreport. See http://www.valgrind.org for details.

GNU Licenses

GNU Licenses

Table of Contents

1. The GNU General Public License 4
2. The GNU Free Documentation License 11

1. The GNU General Public License

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 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 Library 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., 59 Temple Place, Suite 330, Boston, MA  02111-1307  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 Library 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.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as

given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of

the License in the document and put the following copyright and
license notices just after the title page:

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

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.