

Logrind 2

A program trace framework

```
progA.c:1 void _start (void) {
  push  %ebp
  74 (0.0164626): [1] _start (progA.c:1) regwrite esp: bffff550 => bffff54c
  75 (0.0165101): [1] _start (progA.c:1) memwrite bffff54c: bffff55c => 00000000
  mov   %esp,%ebp
  76 (0.0165574): [1] _start + 1 (progA.c:1) regwrite ebp: 00000000 => bffff54c
  sub   $0x8,%esp
  77 (0.0165927): [1] _start + 3 (progA.c:1) regwrite esp: bffff54c => bffff544
progA.c:2 int a = 1;
  movl  $0x1,0xffffffffc(%ebp)
  78 (0.0166320): [1] _start + 6 (progA.c:2) memwrite a: _end + 939227037 => 00000001
progA.c:4 a = 2;
  movl  $0x2,0xffffffffc(%ebp)
  79 (0.0166770): [1] _start + 19 (progA.c:4) jump _start + 19
  80 (0.0167171): [1] _start + 19 (progA.c:4) memwrite a: 00000001 => 00000002
progA.c:6 exit (0);
  sub   $0xc,%esp
  81 (0.0167591): [1] _start + 26 (progA.c:6) regwrite esp: bffff544 => bffff538
  push  $0x0
  82 (0.0167927): [1] _start + 29 (progA.c:6) regwrite esp: bffff538 => bffff534
  83 (0.0168833): [1] _start + 29 (progA.c:6) memwrite bffff534: 00000001 => 00000000
  call  0x8048194
  84 (0.0169495): [1] _start + 31 (progA.c:6) regwrite esp: bffff534 => bffff530
  85 (0.0169896): [1] _start + 31 (progA.c:6) memwrite bffff530: bffff554 => _start + 36
  86 (0.0170452): [1] _start + 31 (progA.c:6) call _end + 133611988
```

MEng Computing
Final Year Individual Project
June 2004

Christopher January
ccj00@doc.ic.ac.uk
chris@atomice.net

Supervisor: Paul Kelly
Second Marker: Olav Beckmann

Abstract

This report describes the design and implementation of a set of tools for capturing and analysing program traces called Logrind 2. The tools are implemented as add-ons to the popular Valgrind and GDB debugging tools. Program traces are captured using dynamic instrumentation and stored in a relational database. Logrind 2 provides support for pretty-printing these traces in a clear and understandable format. The toolset also includes an extended SQL query language that ranges over program traces. Other tools may interface with Logrind 2 using this query language. The program trace schema supports multiple sources and can be used to compare runs of a program. Logrind 2 supports random access navigation of program traces using the concept of a program trace 'cursor'. Users may use this feature to examine the historical state of a process. Program trace capture using Logrind 2 was benchmarked using the BYTEmark benchmark program and the results are described in this report. The report also evaluates the program trace query language with a discussion of possible optimisations.

Contents

1	Introduction	6
1.1	Structure of this report	7
1.2	Contributions	7
1.3	Open source	9
1.4	Example	9
2	Background	12
2.1	Valgrind	12
2.2	The GNU Debugger	12
2.3	Program traces	13
2.3.1	Definition	13
2.3.2	Implementation	14
2.3.3	Applications	14
2.3.4	State of the art	15
2.3.5	Analysing program traces	19
2.4	Program analysis	20
2.4.1	Data flow analysis	20
2.4.2	Control flow analysis	20
2.4.3	Constraint based analysis	20
2.4.4	Dynamic program analysis	20
2.4.5	Dynamic program slicing	20
2.4.6	Conclusion	21
3	Specification	22
3.1	Preliminary Tasks	22
3.2	Interactive debugging support	22
3.3	History command	23
3.4	Support for complex queries	23
3.5	Program trace cursor	24
3.6	Documentation	24
4	Design and Implementation	25
4.1	Tools used	26
4.2	Valgrind	26
4.3	The Logrind skin	28
4.3.1	Example	29
4.4	GDB	30
4.5	The program trace query language	33

4.5.1	Schema	33
4.5.2	Functions	35
4.5.3	Macros	36
4.5.4	Built-in queries	36
4.5.5	Results	39
4.5.6	Limitations	39
4.6	Summary	41
5	Evaluation	42
5.1	Testing	42
5.2	Program trace capture overhead	43
5.3	Query efficiency	47
5.4	Documentation	47
5.5	Open source	48
5.6	What people are saying about Logrind 2	48
5.7	Summary	49
6	Future work	50
6.1	Optimisation	50
6.2	Integration with the main releases of Valgrind and GDB	50
6.3	Better support for multiple sources	50
6.4	Program trace annotations	51
6.5	Graphical user interface	51
6.6	Drill-down on program history	51
6.7	Dependence capture	51
6.8	Macro library	51
6.9	Improved documentation	52
7	Conclusions	53
	Bibliography	54
A	Semantics of program traces generated by Logrind	57
B	Testing schedule	63
C	User guide	73
C.1	What is Logrind 2?	73
C.2	How to install	73
C.3	How to start	73
C.4	Logrind commands	73
C.4.1	logrind history	73
C.4.2	logrind query	74
C.4.3	logind sources	75
C.4.4	logrind macro define	75
C.4.5	logrind macro undefine	75
C.4.6	set logrind cursor	75
C.4.7	show logrind cursor	75
C.4.8	logrind cursor backwards	75
C.4.9	logrind cursor forward	76
C.4.10	set logrind source	76

C.4.11 show logrind source	76
C.4.12 set logrind filter	76
C.4.13 show logrind filter	76
C.4.14 set logrind suppressions	77
C.4.15 show logrind suppressions	77
C.4.16 set logrind max-database-size	77
C.4.17 show logrind max-database-size	77
C.4.18 target logrind	77
C.5 Query syntax	77
C.6 Functions	78
C.7 Suppressions	78

Acknowledgements

Many thanks to Sian Whiting for her loving support.

Thanks to Paul Kelly and Olav Beckmann for their advice, comments and feedback.

Chapter 1

Introduction

In Summer 2003 Jonathon Cooper [Coo03] wrote a skin for Valgrind [vg] (a CPU simulator) called Logrind that logs memory and register writes and jumps made by a program. This log is known as a program trace. Jonathon also extended the GNU debugger [gdb] (GDB) to support program traces; this modified version of GDB is capable of using the program traces generated by Logrind to debug programs *post-mortem* (i.e. after they have exited). Since a program trace contains a complete history of a program's execution it is possible to calculate any of the historical states of the process (proof: appendix A).

This project, Logrind 2, builds on Jonathon's work to add tools for navigating and comprehending the traces generated by Logrind. In particular Logrind 2 supports interactive debugging and querying over program traces. A user may stop a process using the debugger (e.g. by setting a breakpoint), view the program trace captured up to that point and execute a query to find the cause of a problem. Logrind 2 has a brand new architecture to support this new functionality.

Developers often use program traces to assist debugging. For example, manual instrumentation of the program code using print statements is a crude but effective method of obtaining a partial program trace. Logrind eliminates the tedium of manually instrumenting code by using Valgrind to instrument the code automatically and dynamically.

A developer might examine a program trace for:

- Broken constraints (e.g. `ptr != NULL`).
- Unexpected control flow (e.g. caused by miswriting `if (x == 0)` as `if (x = 0)` in C).
- Data corruption (e.g. caused by overwriting the end of an array)

Generally in non-regressive cases, human intervention is required to diagnose a problem because a compiler or debugger cannot know what a programmer's original intention was. However automatic tools can be useful in cutting down the diagnosis time. Logrind 2 allows a developer to issue queries on a program trace that can help them narrow down the search area for a problem.

Program traces also have numerous applications outside of debugging such as profiling and dynamic invariant detection (see Ernst et al. [ECGN01]). Lo-

grind 2 has been designed with such tools in mind by providing a query language built on top of the SQL standard. Other tools can use the query language to retrieve information from the program trace without concern for the specifics of program trace capture.

1.1 Structure of this report

The main achievements of this project are listed below followed by an illustrative example. Then follows a background chapter that introduces the reader to the technologies behind Logrind 2 and looks at the state of the art in program tracing. The remaining chapters follow the evolution of the project, starting with the specification followed by design and implementation. The evaluation chapter explains how the tools were tested and the bugs this testing uncovered. It quantifies the overhead of tracing a program and looks at some problems that can occur with writing queries. The evaluation chapter also assesses what future work will be required to integrate Logrind 2 into the official releases of Valgrind and GDB. The concluding chapters look ahead to the extensive future work this project makes possible and summarise the results of the preceding chapters.

1.2 Contributions

Logrind 2 includes novel functionality that advances the state of the art in program tracing.

- A program trace framework

Overall Logrind 2 provides a framework for building program analysis and debugging tools that depend on program traces. The process of capturing and storing the program trace is handled by Logrind 2: tools need not concern themselves with this. Furthermore the project provides an advanced query language based on the SQL standard that tools can use to access the trace. The query language provides a useful abstraction layer between tools and the program trace itself that isolates the tools from the program trace capture process.

- Interactive debugging of programs running under Valgrind.

The official releases of Valgrind do not support the insertion of breakpoints in the code being simulated, nor do they export the state of the simulated threads. If an attempt is made to debug a program running under Valgrind using GDB, the emulator will abort with an error. The version of Valgrind in the Logrind 2 toolset has been modified to support breakpoints and to export the state of the simulated threads in a shared memory region. The corresponding version of GDB has been modified to read and write register values from the same shared memory region providing interoperability between the two. Similar functionality cannot be found outside of this project.

- Logrind 2 program trace capture memory reads and writes, register reads and writes, jumps, function calls, function returns, system calls and system call results.

Cooper's Logrind skin only captures memory writes, register writes and jumps. The Logrind 2 skin has been extended to capture the additional operations listed above.

- Pretty printing of a program trace with symbol lookup.

When a program trace is captured by Logrind, addresses are left uninterpreted. GDB provides rich symbol table support and this has been utilised to translate uninterpreted addresses into their corresponding symbols. This means function and variable names are displayed in the traces instead of their addresses. Tools built on top of Logrind 2 have access to similar functionality through SQL functions.

- Fast SQL queries over program traces.

The Logrind 2 output file has been changed from a flat file database in the original Logrind to a btree-based one. The database supports SQL queries and is indexed for speed. The SQL language has been augmented with specialised utility functions such as symbol lookup.

- Random access navigation of the program trace.

A 'cursor' may be set anywhere within the program trace. Memory reads, register lookups, etc. are all performed relative to this cursor. This allows the historical state of the process to be examined at any point.

- Multiple program traces in the same database.

The program trace database schema supports multiple traces from different sources. This allows comparison between different runs of the same program or even different programs and supports automated debugging tools that use algorithms such as delta debugging [Zel99].

- Evaluation of program trace capture and query language

The evaluation section of this report quantises the overhead of program trace capture and discusses the significance of this overhead. It examines the efficiency of the query language and suggests some possible optimisations. It also discusses what further work is required for the project aim of an open source release to be achieved.

- Support for C, C++, Objective-C, Java and Pascal

Unlike many program trace tools before it, Logrind 2 is not specific to Java but has support for many different programming languages.

Logrind 2's GDB targets which support program trace queries and interactive debugging for Valgrind are new and original work. No code was used from Cooper's GDB target because Logrind 2 uses a different navigation paradigm, namely program trace queries as opposed to simulated execution.

The core instrumentation code in Logrind 2's Valgrind skin is loosely based on the equivalent code in Cooper's original skin but was extended to trace a larger number of events and to store the events in a relational database.

The database interface module that acts as a bridge between the instrumentation and SQLite is new in Logrind 2.

The total code contribution of this project is about 4300 lines broken down as follows:

Component	Source file	Lines of code
Valgrind skin	valgrind/logrind/logrind.c	853
SQLite bridge	valgrind/logrind/logrindsql.c	329
Logrind target for GDB	insight/gdb/logrind.c	2179
Valgrind target for GDB	insight/gdb/valgrind.c	940
Interactive debugging support for Valgrind	valgrind/coregrind/vg_scheduler.c	100 (approx.)

1.3 Open source

This project aims to produce open source code that will become part of the mainstream release for GDB and Valgrind. It is important that the implementation be readable, well commented and well designed. Furthermore any new functionality must fit into the overall design and goals of the respective projects.

1.4 Example

Figure 1.1 shows the listing for `example.c`.

Let us say that a thread is only allowed to write to the variable `shared_variable` if it holds a mutex. Logrind can be used to check this constraint is enforced. Assuming the source file has been compiled to an executable called `example`, the user invokes GDB in the following way:

```
gdb ./example
```

To run the program under Valgrind the user types these commands:

```
target logrind
run
```

After a short while the gdb prompt will return. The user may then issue the following command to check the constraint:

```
logrind history data = symbol2beginaddr(sequence, "shared_variable")
and sequence not in (select write.sequence from Memory-
Write write, Jump lock, Jump unlock, Jump unlock2 where
lock.address = symbol2beginaddr(lock.sequence, "pthread_mutex_lock")
and unlock.address = symbol2beginaddr(unlock.sequence,
"pthread_mutex_unlock") and unlock2.address = symbol2beginaddr(unlock2.sequence,
"pthread_mutex_unlock") and write.address = symbol2beginaddr(write.sequence,
"shared_variable") and lock.sequence < unlock.sequence and
lock.sequence < unlock2.sequence and lock.sequence < write.sequence
and write.sequence < unlock.sequence and lock.thread = un-
lock.thread and lock.thread = unlock2.thread and lock.thread
= write.thread group by write.sequence,lock.sequence,unlock.sequence
having unlock.sequence=min(unlock2.sequence))
```

Although the query looks complicated it is simply long rather than complex. The sub-query finds all the writes to `shared_variable` that are sandwiched by a `pthread_mutex_lock` and `pthread_mutex_unlock` pair. Any writes

that do not occur in the results of this sub-query are writes outside of a mutex. The having `unlock.sequence=min(unlock2.sequence)` portion checks that the `pthread_mutex_unlock` call comes directly after the lock.

In practice common queries would be defined as macros so the user does not have to write them from scratch each time. There is scope for improving the query language's support for sequences and this is discussed in section 4.5.6.

The observant reader may notice that the calls to `pthread_mutex_lock`, etc. are jumps rather than function calls. This is because functions in shared objects are called indirectly through a jump table.

When the above query is issued it returns one result:

```
example.c:14 shared_variable = 2;
movl    $0x2,0x8049664
211 (0.0308530): [3] bad_thread + 3 (example.c:14)
memwrite shared_variable: 00000001 => 00000002
```

This tells us that thread 3 (as denoted by [3]) wrote to `shared_variable` outside of a mutex in line 14 of `example.c`.

Figure 1.1: example.c

Chapter 2

Background

This chapter introduces the technologies used by Logrind 2, namely Valgrind, the GNU Debugger and program tracing. A sample of papers representing the current state of the art in program tracing have been summarised to show how it is currently being used.

2.1 Valgrind

Valgrind is a platform for dynamic program instrumentation written by Julian Steward. It supports different skins that specify the instrumentation that will be applied to a program run using Valgrind. Examples of Valgrind skins include memcheck, a 'skin that detects memory-management problems in your programs.'; cachegrind, a cache profiler; and helgrind, 'designed to find data races in multi-threaded programs.' Valgrind is licensed under the GNU General Public licence [\[gpl\]](#).

Valgrind's core is a shared library (`valgrind.so`) which is loaded before any others by the dynamic linker (`ld-linux.so`). Valgrind provides a synthetic CPU which runs the same code as the original program with extra code added in between (the instrumentation) which provides services not provided by the real CPU (such as bounds checking). The process of adding the instrumentation to the original code is called translation. Code is translated in units called basic blocks (a block of code without jumps or returns) and each translated block stored in a cache. Valgrind performs just-in-time instrumentation. When the real CPU enters `valgrind.so`'s initialisation code, Valgrind interrupts the normal execution of the process and execution continues on the synthetic CPU provided by Valgrind. When the main program has finished the synthetic CPU will reach `valgrind.so`'s finalisation code. At this point Valgrind will halt the synthetic CPU, do any final checks and then return control to the real CPU which will `exit()` the process.

2.2 The GNU Debugger

The GNU Debugger (GDB) is a symbolic debugger that can debug programs written in 'C, C++, Pascal, Objective-C (and may other languages)'. GDB is

part of the GNU Project [[gnu](#)] founded by Richard Stallman and as such is licensed under the GNU General Public licence. The GDB documentation states ‘The purpose of a debugger such as GDB is to allow you to see what is going on ‘inside’ another program while it executes—or what another program was doing at the moment it crashed.’

GDB allows a user to:

- Control the invocation of a program
For example, GDB allows a user to specify the command line arguments of a program.
- Control the execution of a program
For example, by executing one instruction at a time (single step).
- Stop the program when particular conditions are satisfied
For example, by specifying a breakpoint on a source line.
- Examine the state of a program
For example, by evaluating a simple expression containing program variables.
- Modify the state of a program
For example, by changing the value of a variable.

GDB uses a command-line interface however there are a number of graphical user interfaces available for GDB including Insight [[ins](#)] and the Data Display Debugger [[ddd](#)].

2.3 Program traces

2.3.1 Definition

A program trace is the set of process traces of each process in a given program. A process trace captures the sequence of states traversed by a process from initialisation to termination [[FH93](#)]. It may be shown that there are a number of equivalent definitions of program state. For example in certain programming languages it is possible to show that the mapping of variables to values and the contents of a process’s memory are equivalent definitions of program state. This equivalence does not hold true, however, for languages with dynamic storage allocation such as C. We may define formal semantics for program traces just like programming languages. These semantics are useful for reasoning over program traces and performing program analysis [[CL96](#)]. Appendix [A](#) defines the operational semantics for the program traces generated by Logrind.

It is usually more efficient to store the transitions between process states in a program trace rather than the states themselves. If a program trace is complete and the initial state is known then this is equivalent to storing the states themselves (proof: appendix [A](#), [[CL96](#)]). For a native process the transitions between states are memory and register writes.

2.3.2 Implementation

Capturing a program trace is often difficult as few programming languages have support for generating traces built-in. It is possible to modify the compiler to output trace information but this requires existing programs to be re-compiled before a program trace can be captured [ADS93]. There are other solutions, however, that allow pre-compiled programs to be traced.

One such solution is to single step through a program recording writes as they occur. The Spyder debugging tool [ADS93] uses transparent breakpoints in an analogous way; after each breakpoint control is transferred to the debugger which updates the execution history of the program and then resumes the original program. However Spyder has a context switching overhead associated with transferring control to the debugger and back when one of the transparent breakpoints occur.

An alternative to single stepping through the program is to use instrumentation [CFC01]. Here the original program is 'patched' to record memory writes and other trace information; the program does not need to be re-compiled. Static instrumentation tools generate a modified executable that generates trace information and is run as normal. Dynamic instrumentation tools such as Valgrind add instrumentation to a program at runtime using a just-in-time translator.

Usually tools to add program instrumentation are coded by hand but it is possible to automate the process. For example the Wyong tool uses the Eli [Slo97] compiler generation system and the ATOM program generation system to allow dynamic program analysis tools to be automatically built from an Eli specification. Goldsmith [SGA04] describes a method of using relational queries over program traces to drive instrumentation.

2.3.3 Applications

Historical state inspection

Program traces provide for detailed inspection of a program's state at a particular point in its execution history without requiring the program to be re-run (presumably with debugging output added). This follows from the definition of a program trace.

Reversible debugging

Reverse execution is the ability to backtrack, or retrace, the execution of a computer program and is a particularly useful feature of a symbolic debugger [Zel73, CFC01]. Debugging with reverse execution is known as reversible debugging. Reverse execution of a program may use checkpointing, program traces, or reverse computation. Often a combination of the three may be the most efficient solution.

Using a reversible debugger a user may go back to an earlier point in a program's execution history, modify a variable or two and then restart the program from that point to see if it fixes a bug. However in practice it is usually unnecessary to restart the program, merely viewing the historical state of the program is enough to diagnose a problem.

Dynamic program slicing

See [2.4.4](#).

Dynamic query-based debugging

A query-based debugger allows a user to test variable and object relationships using specially formulated queries [LHS97]. Such queries may be run when a conditional breakpoint is reached, for example, and can check that data structures are correct. However it is expensive to re-evaluate a query in its entirety every time such a breakpoint is reached. Dynamic query-based debugging overcomes this problem by re-evaluating the query incrementally each time a variable or field that could affect the result of the query changes [LHS99]. In post-mortem debugging incremental re-evaluation can be used to efficiently evaluate queries that may range over large parts of the program trace.

Automated debugging

Automated debugging attempts to automate some (or all) of the processes humans use to debug a program. An example of automated debugging is delta debugging which looks at those statements that are executed on a successful run of a program and those statements that are executed on a failed run and concludes the bug must exist in the difference between the two sets of statements. The set of statements that were executed can be obtained from a program trace. Further test cases can be used to narrow down the candidate set which a human can then inspect and correct. Another related technique that uses program traces is critical slicing [Hsi93]; critical slicing uses heuristics to identify suspect statements from success and failure cases.

Performance analysis

A program may be traced running on one system with real I/O and then the trace replayed on another system with emulated I/O to compare the performance of the two systems.

Invariant detection

A novel use of program traces is to detect invariants for functions and methods. The Daikon [ECGN01] invariant detector uses program traces generated by front-ends for C, C++, Java and other languages to find invariants amongst complex data structures including conditional invariants.

2.3.4 State of the art

As processor speeds increase the relative overhead of program tracing is reduced. This means that program tracing becomes more practical as time goes by. A number of papers on program tracing have been published in recent years. Two new tools built using Valgrind were published just over the lifetime of this project. Presented below is a representative sample of products and papers that demonstrate the current state of the art in program tracing.

Detailed Program Tracing and Replaying: Theory and Implementation

Jonathon Cooper [Coo03] implemented a skin for Valgrind called Logrind that adds instrumentation for three different events: memory writes, register writes and jumps. Logrind records details of these events in a trace file whenever they occur in a program. Jonathon claims a debugging tool can reproduce the value of a program's variables at a particular point in its execution history by analysing the program trace generated by Logrind. He modified the GNU Debugger to allow it to work on the traces produced by his Valgrind skin. His implementation was a prototype that shows it is possible to modify Valgrind and GDB to support program traces. Logrind does not record memory or register reads and therefore cannot be used for dynamic program slicing although Jonathon claims it 'can easily be modified' to do so. Jonathon's Valgrind skin does not permit a debugger to interact with the program being traced while it is running. Rather the program may only be debugged post-mortem using the program trace.

Debugging Backwards in Time

Lewis [Lew03] introduces the concept of an omniscient debugger that 'works by collecting events at every state change ... and every method call in a program.' An omniscient debugger allows the programmer to 'look at the state of the program at any time desired'. It is 'possible to 'single step' the program forwards or backwards, to step to any method call, follow an exception throw, any context switch, etc.'

Lewis provides a concrete implementation of an omniscient debugger for Java called the ODB [ODB]. The ODB 'collects information by instrumenting the byte code of the target program as it's loaded.' Lewis' paper concentrates on the 'presentation of information and 'navigation' ' rather than the details of the instrumentation of the Java byte code.

ODB does not permit interaction with a program while it is running although it does allow the program trace logging to be stopped before a program has finished executing.

Retrovue

Retrovue [Ret] is a commercial Java debugger billed as 'The Total Recall Debugger'. The debugger 'keeps track of every operation executed by your program, allowing you to 'roll back the clock' to any previous point in time and examine the entire state of your program as it was at that instant.' Visiomp claim Retrovue enables the user to: 'determine when a variable was last modified; find the cause of unexpected exceptions; trace bugs back to their source; see complete state at any previous moment; review all previously executed operations; find memory leaks; analyse captured journals of elusive bugs; step back through or rewind execution; eliminate the need to restart; determine the cause of deadlocks; understand thread behaviour; unravel thread interactions' and 'perceive deviations from expected patterns'.

Retrovue keeps a log of method calls and return values, field assignments and thread context switches which are displayed in the 'History View'. Logrind 2 implements a similar view. The debugging supports simple searches

on this program trace but does not appear to support more complex queries. Retrovue supports 'standard' debugging functions in addition to its more advanced features including breakpoints and watches. Unlike Cooper's Logrind, Retrovue can interact with a program while it is still running. Logrind 2 also provides this functionality.

The main limitation of Retrovue is that it only supports Java code running in a Java Virtual Machine.

Jinsight/WebSphere Studio Application Developer (JSAS)

Jinsight [Jin] was an IBM research project that developed technology now included in IBM's commercial Java IDE. It uses instrumentation of Java byte code to obtain program traces which are used to visualise Java programs. 'Visualisations let you understand object usage and garbage collection, and the sequence of activity in each thread, all from an object-oriented perspective.' Jinsight provides similar functionality to Retrovue, for example: pattern analysis, data views, memory and time profiling and memory leak diagnosis. Unlike Retrovue, Jinsight doesn't act as a live debugger but does provide a command console which can be used to stop and start tracing.

Trace-Based Debugging

Reiss [Rei93] describes a method for collecting program traces that is very similar to that employed by Cooper. He suggests accumulating control flow information by logging entry to basic blocks and tracing memory information by recording memory writes. Reiss asserts that recording every register write is too costly and instead suggests only recording the contents of registers at the start of a basic block. Later he suggests that the format the program trace is presented to the user in be based on basic blocks rather than individual events.

Reiss suggests using a shared memory buffer to communicate between the trace capture program and the debugger allowing interactive debugging to take place whilst the trace is being captured. Such a mechanism was implemented in Logrind 2.

Reiss goes on to mention some of the problems with tools such as Logrind:

1. The overhead of capturing the trace data slows down the program being traced 'from a factor of ten to one hundred depending on the application.'

Reiss suggests optimising the code could cut down this overhead considerably and this could yield speed-ups in Logrind.

2. The large amount of data gathered; two gigabytes of disk space is sufficient for 'twenty seconds of real execution time'.

Reiss suggests compression as one way to work round this limitation as well as filtering the data gathered (e.g. not recording events that occur inside standard shared libraries.)

3. Providing some meaningful way of accessing the data.

A program trace is large and rather unwieldy and almost impossible to use without some assistance.

Almost: Exploring Program Traces

Almost [RR99] is a visualisation tool for program traces. It is similar in functionality to Jinsight. Program traces are collected using the method described in Reiss's paper above. The tool provides both linear and spiral views of function calls. The visualisations are linked to the original code which may be called up at the press of a button. The tool also provides for visual data mining by colouring according to properties such as time of first activation of a function.

Debugging with Dynamic Slicing and Backtracking

Agrawal, DeMillo and Spafford [ADS93] present a debugging tool called Spyder. Spyder uses a modified version of GDB and transparent breakpoints to capture program trace information. The tool is capable of calculating dynamic program slices and execution backtracking.

The features in Spyder overlap in some ways with the features that are implemented in Logrind 2. The main difference between Spyder and Logrind 2 is that Logrind 2 uses dynamic instrumentation instead of transparent breakpoints to record the program trace thus eliminating the context switching overhead in Spyder. In addition, this project provides an explicit visualisation of the execution history of a program along with the ability to perform advanced queries.

Reversible Debugging Using Program Instrumentation

Chen, Fuchs and Chung [CFC01] suggest implementing a reversible debugger using program instrumentation. The original program A and the instrumented version of the program $A_{instrument}$ run in separate processes, only one of which may be running at any given time. The user can turn on trace recording by switching control from A to $A_{instrument}$ and vice versa. Instrumentation can occur on more than one level: source code, assembly code and machine code levels but the authors suggest that machine code instrumentation makes for easiest automation since the exact effects of a machine code instruction are known.

The authors suggest storing the execution history in a wraparound history buffer thus limiting the maximum size of the history. Although such a limit may seem restrictive, it is often the tail end of the history near to the point of failure that is of most interest to the user. This feature is implemented in Logrind 2.

Redux: A Dynamic Dataflow Tracer

Redux [NM03], like Logrind, uses Valgrind to dynamically instrument programs. Redux 'records the dataflow — inputs and outputs — of every operation that produces a value' [NM03]. Redux computes DDFGs (dynamic dataflow graphs): directed acyclic graphs which show the dependency between values in a process. Unlike Logrind 2, Redux traces reads and write to Valgrind's internal temporary registers (which are used to hold intermediate values) as well as the emulated architecture registers (such as `eax`). This

allows Redux to generate full dependency information. The overhead of instrumenting operations involving Valgrind's temporary registers is very high so Logrind 2 does not instrument these registers. This means it is not possible to generate dependency graphs from the program traces captured using Logrind 2. Nethercote gives applications for DDFGs in debugging and dynamic program slicing. Unfortunately Redux does not scale well since the resulting DDFGs are highly connected and very dense. Nevertheless the implementation ideas used in Redux could be adapted to add full dependency analysis support to Logrind 2.

Kvasir

Kvasir is a front end for Daikon [Dai], an invariant detector. Kvasir records function calls (including their arguments) and function returns (including the return value). Kvasir also has limited support for references and pointers, i.e. it will also record pointed to values if appropriate, to a limited depth. Unlike Logrind 2, Kvasir performs symbol lookup at capture time rather than analysis time. Kvasir cannot be used to reconstruct a program's historical state because it does not store enough information. Specifically it does not record the value of global variables — data stored in global variables might never be passed as a function parameter/return value and so not appear in Kvasir's output.

2.3.5 Analysing program traces

A program trace of any non-trivial program will generally be very long and contain a large amount of data [Rei93]. It is impractical to inspect a raw program trace manually. Instead program trace tools offer a query language to allow the user to inspect an interesting portion of the program trace at a time and/or visualisation tools that allow the user to take in more information by presenting it visually.

There are many ways of presenting program traces visually. A program trace could be presented as a control flow or data flow diagram, for example. Almost [RR99] has the capability of presenting function calls in a novel spiral format. In practice it would take too long to layout a large program trace in its entirety and the result would be too dense to be comprehensible. Program trace visualisation tools, therefore, will generally only show a portion of the program trace at any given time. The portion of the program trace on display is controlled using a filter. Since a filter is a specialised form of query it is clear that a well designed query language is an important components of any program trace analysis tool.

Relational algebra [Cod83] is an established formalism for querying large bodies of data. SQL [sq192] is a database query language based on relational algebra. SQL can be used to query program traces directly but requires explicit knowledge of the program trace schema. Program Trace Query Language (PTQL) [SGA04] is an example of a language based on SQL explicitly designed for writing declarative queries over program traces. PTQL specifies a standardised program trace schema but tools may use a different internal representation.

Aspect Oriented Programming introduces the notion of a join point defined as 'certain well-defined points in the execution of a program' [KHH⁺01]. Ex-

amples of a join point include a method call and a variable assignment. A pointcut is a set of join points and may be specified using a pointcut designator. Join point specifications would seem to offer a natural way of specifying dynamic program slices (see 2.4.5) that might be useful in debugging or program analysis.

2.4 Program analysis

'Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer.' [NNH99]. Program trace analysis is really a specialised form of dynamic program analysis. There are a number of different types of program analysis:

2.4.1 Data flow analysis

Data flow analysis determines the data dependencies in a program (one variable's value may depend on the value of another as in the statement $y := x + 1$). In order to calculate the data flow in an imperative program it is also necessary to know the control flow.

2.4.2 Control flow analysis

Control flow analysis determines the possible control paths between basic blocks in a program and the conditions under which each path will be executed.

2.4.3 Constraint based analysis

Constraint based analysis generates a collection of constraints from a program which are used to determine possible control flows. The least solution of these constraints is used to generate an approximation of the control flow graph. In languages with unconstrained pointers, such as C, it becomes difficult to generate a tight set of constraints.

2.4.4 Dynamic program analysis

The definition of program analysis above defines static program analysis, i.e. analysis performed on a program's source code that ranges over all possible execution paths. However there is another form of program analysis called dynamic program analysis that ranges over a particular execution path, perhaps while a program is running. An interactive debugger such as GDB may provide dynamic program analysis tools that can be used to analyse a program while it is suspended.

2.4.5 Dynamic program slicing

Program slicing is a form of data flow analysis. Weiser [Wei81] defined a program slice S as 'a reduced, executable program obtained from a program P by removing statements, such that S replicates part of the behaviour of P '. A

more useful definition of a program slice is a subset of the statements of a program that directly or indirectly affect the value computed at a particular point in the program (forward program slice) or are directly or indirectly affected by said value (backward program slice).

Static program slices are calculated using static program analysis when the real execution path of a program isn't known. Often an exact slice is incomputable since the constraints used to determine the control flow are themselves incomputable; a superset of the program slice may be calculated instead [Tip94]. Dynamic program slices, on the other hand, are always computable because the actual control flow is known. Therefore static program slices tend to range over approximately all possible execution paths, whereas a dynamic program slice ranges over a single known execution path. A program slice computed using a program trace is a dynamic program slice since the program trace specifies the actual execution path of the program.

Note if a program is being debugged interactively while a program trace is being recorded then only the backwards dynamic program slice is calculable from the program trace; the forward program slice must be calculated statically since no future program trace is available. However if a program is being debugged post-mortem then the whole program trace is available to the debugger so exact forward slices may be calculated as well.

2.4.6 Conclusion

Program traces are being used in many different ways including automated debugging, assisted debugging, invariant detection, performance analysis and reversible debugging. Program traces are versatile and therefore a tool built upon them should not be artificially restricted in scope. A single tool could not hope to offer functionality in every area listed above but Logrind 2 provides a framework that specialised tools may be built upon. Logrind 2's rich query language built upon the established SQL standard makes it easy to write these tools. Many people have independently conducted work with program traces that lead to similar results. Logrind 2 provides a foundation that will allow the state of the art to be advanced by providing a platform for new tools to be built upon.

Chapter 3

Specification

This informal specification was part of the design process for Logrind 2 and guided the implementation. Cues were taken from the existing commercial and non-commercial tools that use program traces for debugging and program analysis to choose which features to implement. Most of the existing implementations given in section 2.3.4 offer some kind of pretty printing and/or 'history view' functionality. Spyder offers dynamic program slicing and reverse execution. The two commercial products (Retrovue and JSAS) offer interactive debugging. However it was important for this project to also offer something novel. We wanted to provide a framework that would enable all the foregoing tools and more to be written without concerning themselves with the details of program trace capture. To this end Logrind 2 provides an efficient mechanism for performing complex queries on the execution history of a program that isolate tools from the underlying capture process.

This section outlines the functionality that was added to Logrind 2 with the appropriate specifications. Section 5.1 evaluates how well the implementation matched the specification.

3.1 Preliminary Tasks

Before any new functionality was added to Logrind 2, Cooper's original code needed to be updated to the latest versions of GDB and Valgrind. At the start of the project these were 6.0 and 2.0 respectively. Since then new version of Valgrind and GDB have been released. The new version of Valgrind is fundamentally incompatible with the changes made to support interactive debugging support so Logrind 2 continues to use the older version. The new version of GDB (6.1) offers no compatibility problems and it should be trivial to update the implementation to use this version.

3.2 Interactive debugging support

Interactive debugging allows a user to interact with the process being run under Logrind 2 as it is executing from within the debugger. Valgrind does not support interactive debugging by default so this is new functionality.

1. The debugger must be enhanced to provide an option for launching the inferior under Logrind 2.
2. The debugger must make any commands that work on program traces available for use if a process is launched under Logrind 2 in this manner.
3. The debugger must not emulate standard GDB commands that affect execution such as `step` and `cont`).
4. The debugger should allow the user to set the maximum size of the program trace.

3.3 History command

The history command shows assignments carried out by the program, function calls and system calls. The command is integrated with the command line GDB. The program trace is pretty printed, making full use of visual effects (colour, indentation, etc.) to assist comprehension. An example of a pretty printed program trace can be found in [RR99].

1. The history command must be implemented as a GDB command.
2. The history command should show writes made by the program, jumps, function calls, function returns and system calls and the source line where these occurred. If read events are present in the program trace, the history command should show these as well.
3. The history command should also show the index of the relevant program trace event.
4. The history command should translate addresses to variable or function names whenever possible.
5. The history command should abort with an error if the debugger is not using a program trace vector.
6. The history command should allow a user to specify only a portion of the program trace to display.

3.4 Support for complex queries

The history of a program can be long and unwieldy. This deliverable adds functionality to the debugger that allows a user to filter the output of the history command as well as performing more complex queries.

1. A filter or query may contain:
 - boolean expressions in the language of the program being debugged (e.g. `x > 5`)
 - comparisons between one or more elements from the program trace schema (e.g. `opcode = opcode_jump`)

- functions of elements from the program trace schema as comparison operands (e.g. `addr2line(pc) = "main.c:5"`)
joined by logical connectives (AND, OR, etc.)
2. The history command should apply a user-specified filter to its output.
 3. The history command should only output events that match the filter.

3.5 Program trace cursor

The program trace cursor allows a user to inspect the historical state of a process. The cursor command allows the user to set the cursor.

1. The cursor command should set the program trace cursor to the given location.
2. The cursor command should allow the user to turn the program trace cursor off (memory, registers and stack are read from the inferior as usual).
3. The debugger should return values for memory, registers and stack that they had at the cursor position.

3.6 Documentation

Each deliverable should also include appropriate developer and user documentation. Developer documentation includes source code comments and additions to documentation of Valgrind and GDB's internals. User documentation should either be added to the user guide for the respective programs or as a separate unified user guide for Logrind 2.

Chapter 4

Design and Implementation

This chapter discusses the design issues that were important in the development of Logrind 2 as well as giving details of some of the more interesting aspects of the implementation.

Logrind 2 has three components:

1. A modified version of Valgrind, the CPU simulator.
2. A modified version of GDB, the GNU debugger.
3. A skin for Valgrind called Logrind 2.

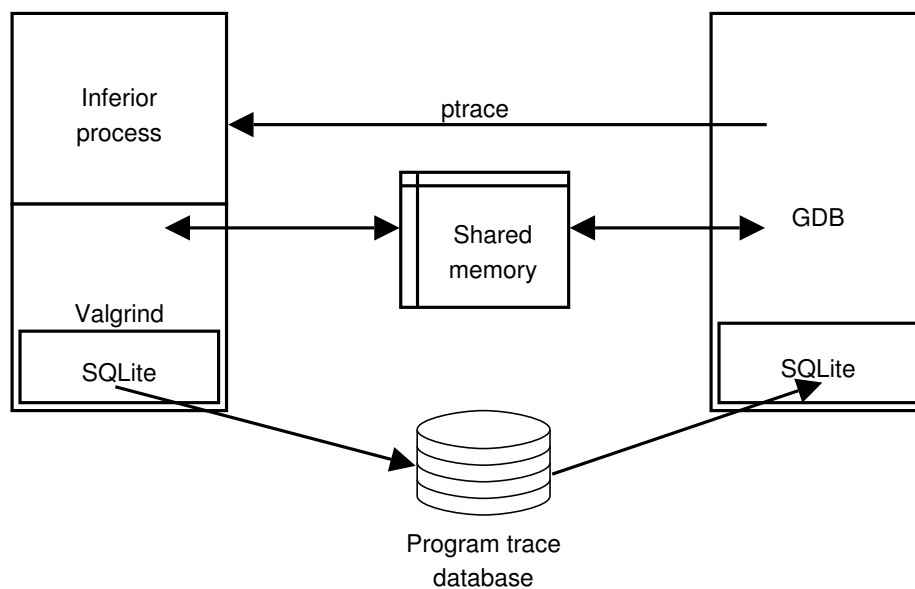


Figure 4.1: High-level architecture of Logrind 2

Figure 4.1 shows how these components interact. When GDB starts the inferior process it calls the `ptrace` system call to indicate to the kernel that the process is to be traced by its parent — GDB. Whenever a signal is raised in the

inferior process the process is stopped and GDB notified. GDB may examine the state of the inferior process using `ptrace`.

However when the inferior process is running under the Valgrind simulator the state of the inferior process that GDB sees will be the state of the simulator rather than the simulated process. For this reason Logrind 2 stores the state of the simulated process in a shared memory region. When GDB is told to debug the simulated process it reads register values from this shared memory region instead of using the `ptrace` call.

The Logrind 2 skin for Valgrind dynamically instruments the inferior process' code to record memory writes, jumps, function calls, etc. These events are stored in a database using the SQLite embedded database engine. GDB allows the user to query the program trace using the same embedded database engine. The database is only accessed by one process at any given time so there are no locking issues involved. This is enforced using a synchronisation variable in the shared memory region.

4.1 Tools used

The implementation was typed in KDevelop, the KDE integrated development environment, and vim. GCC was used to compile the files. The latest version of `binutils` was used to strip unnecessary symbols from the Valgrind skin. Older version don't have the necessary `-wildcard` option. The GNU debugger was used to debug the skin and GDB itself.

This report was typed in the Textpad text editor and typeset using the Cygwin version of LaTeX.

4.2 Valgrind

Valgrind does not include support for debuggers. If a user tries to debug a program running under Valgrind using GDB, Valgrind will abort with the message `Unexpected Grp3 opcode`. The error occurs because the breakpoint machine code instruction (`CCh` or `INT $3`) is not recognised by the emulator.

Logrind 2 adds support for breakpoints to Valgrind. Breakpoints are treated much like system calls in that they end a basic block of straight line code. This causes control to return to the dispatcher, a Valgrind function responsible for handling system calls and translation cache misses (the translation cache contains the basic blocks translated so far). The dispatcher will look at a thread return code (TRC) that tells it why the basic block was exited. When the basic block is ended because of a breakpoint the TRC will contain the value `VG_TRC_EBP_JMP_BREAKPOINT`. The dispatcher will respond by raising a `SIGTRAP` signal. This signal is the same one raised by a breakpoint interrupt.

Unfortunately it is not sufficient to simply raise the correct signal when a breakpoint is encountered as the program counter where the signal is raised will be different from where the breakpoint was set. A debugger will not interpret the signal as caused by a breakpoint, but rather by some other cause (`SIGTRAP` can be raised when a user mode process attempts to use a privileged instruction, for example). The debugger requires access to the simulated CPU's program counter to recognise breakpoints correctly.

There are three ways in which the debugger might obtain the state of the simulated process.

1. Using message passing IPC

Add explicit support to both Valgrind and the debugger to allow them to communicate using some message passing IPC mechanism such as Unix domain sockets. The Valgrind process would act as a server and the debugger would send command messages to the server to retrieve registers, etc.

The advantage of this method is that it isolates Valgrind from the debugger. The disadvantage is that it requires a remote debugging server to be implemented as part of Valgrind.

2. Using GDB remote debugging support

GDB supports remote debugging over TCP/IP. There are program stubs available which implement the server side of this protocol (GDB implements the client side). One of these stubs could be linked with Valgrind to add debugging support.

The advantage of this method is that, as above, it isolates Valgrind from the debugger. In particular no modifications need to be made to the debugger. There will also be less to implement since it uses a standard protocol that is already defined and implemented. The disadvantage with this method is that the protocol is not extendable to support program trace specific features.

3. Using a shared memory region

The state of the threads emulated by Valgrind could be exposed in a shared memory region as suggested by Reiss [Rei93]. The debugger could map the same shared memory region in its process to access the state of the inferior's threads.

The advantage of this method is that it is easy to implement. The disadvantage is that the exposed data structures may be fragile (e.g. if they are internal data structures) preventing forward and backward compatibility.

Logrind 2 implements the third option — using a shared memory region — because i) there was a risk that both of the other options may have taken a long time to implement and ii) support for interactive debugging is not the main focus of the project. A patch that implements the shared memory region solution was sent to the Valgrind mailing list. From the responses it is clear that that implementing a server for the GDB remote debugging protocol in Valgrind would have been the most forward compatible solution and the solution most acceptable to the Valgrind developers. It also has the advantage that no modifications need to be made to GDB. Unfortunately there was not enough time to implement these changes.

The version of Valgrind in the Logrind 2 toolset creates a shared memory region when it initialises. When a debugger attaches to the process it can use the process ID of the inferior to open the same shared memory region since the shared memory key is a function of the process ID. The shared memory contains an array of structures that contain information about the threads running

on the simulated CPU (the thread block). Of particular interest to a debugger is the set of registers stored for each thread. The thread block is updated every time control returns to the Valgrind dispatcher, including when a breakpoint is encountered in the emulated code. The index number of the thread that is currently running is also stored in the shared memory region. When a signal is raised by the inferior a debugger can read the program counter for the current thread from the thread block to determine where in the simulated code the exception occurred. In this way a debugger can match SIGTRAP exceptions raised by the inferior to the corresponding breakpoints.

If a signal is raised for any reason other than a breakpoint (for example the user presses Ctrl-C) then the Logrind 2 skin will have an active write transaction on the program trace database. GDB will be unable to obtain a read lock on the database because SQLite does not support nested transactions. To work around this limitation whenever a signal occurs the Logrind 2 GDB target will resume the inferior process and ask the skin to end the active transaction by means of a synchronisation flag in the shared memory region. When the inferior process has noticed the flag and committed the transaction it will raise a SIGTRAP signal to inform the debugger it has reached a safe point. Signals retain the correct semantics because the operating system is not obliged to immediately deliver a signal. SIGSEGV and SIGBUS signals cause Valgrind to immediately stop executing client code so the synchronisation flag will be noticed straight away in the case of these signals.

One of the problems with program tracing is the vast amount of data generated. Reiss [Rei93] suggests filtering the data gathered to reduce the amount of data. Valgrind supports suppressions which are used to suppress known errors in, for example, system libraries. Logrind 2 extends this mechanism to program traces. By default Logrind 2 will not trace system libraries or the Valgrind library itself.

4.3 The Logrind skin

The Logrind 2 toolset contains a skin for Valgrind called Logrind that captures program traces. Valgrind decompiles IA32 machine code to a microcode language called UCode. Each basic block of UCode is passed to the skin for instrumentation. The skin examines each UCode instruction and instruments them as appropriate.

The following code fragment shows how memory writes are instrumented by Logrind 2:

```
case STORE:
    VG_(set_global_var) (cb, (Addr) & opcode,
        trace_opcode_memwrite);
    VG_(set_global_var) (cb, (Addr) & pc, x86_instr_addr);
    VG_(set_global_var) (cb, (Addr) & imageLen, u_in->size);
    if (u_in->tag2 == TempReg)
    {
        t_addr = u_in->val2;
    }
    else
```

```

    {
        sk_assert (u_in->tag2 == RealReg);
        t_addr = newTemp (cb);
        uInstr2 (cb, MOV, 4, RealReg, u_in->val2, TempReg, t_addr);
    }
VG_ (set_global_var_tempreg) (cb, (Addr) & data1, t_addr);
copy_mem (cb, preImage, t_addr, u_in->size);

VG_ (copy_UInstr) (cb, u_in);

VG_ (call_helper_0_0) (cb, (Addr) & trace_write_mem_event);

```

The `set_global_var` function sets a global variable to a value calculated at instrumentation time — it will be constant at run-time. These values include things such as the program trace opcode `trace_opcode_memwrite` and the program counter.

The `set_global_var_tempreg` function is used to set a global variable to a value stored in a temporary register at run-time. In the case of a memory write this function is used to store the address being written to in a global variable.

The `call_helper_0_0` is used to add UCode that calls the `trace_write_mem_event` skin function. This function writes the information stored in the aforementioned global variables to the program trace.

The `copy_UInstr` function adds the original UCode instruction to the instrumented code.

It is necessary to store the information about each operation in a global variable because the instrumented code does not share a stack or heap with the skin.

4.3.1 Example

Figure 4.2: UCode before instrumentation

```

0x804823B:  addl %edx, (%eax)

    9: GETL      %EAX, t10
   10: LDL      (t10), t12
   11: ADDL     %EDX, t12 (-wOSZACP)
   12: STL      t12, (t10)
   13: INCEIPo  $2

```

Figure 4.2 shows the UCode generated for an IA32 add instruction. Figure 4.3 shows the same UCode after being instrumented by the Logrind skin.

Lines 12 to 20 are inserted by `set_global_var` (see above). Lines 21 and 22 store the address being written to in a global variable. The `t10` register contains the contents of the emulated EAX register. Line 23 is the call to `trace_write_mem_event` that writes the event to the program trace. The

Figure 4.3: UCode after instrumentation

```
9: GETL      %EAX, t10
10: LDL      (t10), t12
11: ADDL     %EDX, t12 (-wOSZACP)
12: MOVL     $0x1, t16
13: MOVL     $0x4001A208, t14
14: STL      t16, (t14)
15: MOVL     $0x804823B, t20
16: MOVL     $0x4001A20C, t18
17: STL      t20, (t18)
18: MOVL     $0x4, t24
19: MOVL     $0x4001A2A0, t22
20: STL      t24, (t22)
21: MOVL     $0x4001A210, t26
22: STL      t10, (t26)
23: CCALLo   0x40017560()
24: STL      t12, (t10)
25: MOVL     $0x2, t30
26: MOVL     $0x4001A208, t28
27: STL      t30, (t28)
28: CCALLo   0x40017560()
29: INCEIPo  $2
```

first call of the function stores the pre-image of the memory write (i.e. the contents of the memory before it was written to). The second call (line 28) stores the post-image (i.e. the contents after writing). In this case the post-image will be the contents of the t12 register which holds the result of adding EDX to the contents of the memory pointed to by EAX.

4.4 GDB

The Logrind 2 toolset contains a modified version of GDB that adds support for i) processes running under Valgrind and ii) program traces.

The design and implementation decisions associated with adding Valgrind support to GDB are described in section 4.2 so this section will concentrate on the support for program traces that was added to GDB.

Cooper's Logrind skin stores the program trace in a flat file. He modified GDB to support post-mortem debugging using this program trace. Whereas GDB would normally read registers and memory from the inferior process, Cooper's modified GDB supplies register and memory values from the program trace. This is achieved by replaying the trace in place of the real execution of the inferior with the added advantage that the trace can also be replayed backwards. When a user types run at the GDB prompt Cooper's code starts reading the program trace and pretends to execute the same instructions the original process did. This has the advantage that it doesn't require any changes to GDB's process model. Furthermore familiar debugging methods,

such as setting breakpoints, can be used with the program trace. However the main disadvantage of this method is it does not afford the user the full benefit of a program trace. The user may only view the program trace through a narrow window — the current point in the simulated execution.

Logrind 2 eschews the simulated execution model for a query language. A user may issue queries against the program trace to assist them in debugging a program. This allow the user to take full advantage of the program trace because the whole trace is available to the user at any time. On the other hand if the user wishes to see the process state at a particular point in its history as Cooper's code allowed, that is still supported with the added advantage that positioning in the trace is now random access rather than sequential.

A flat file format could never support queries and random access to the program trace efficiently since entry lookup is always $O(n)$. Logrind 2, therefore, replaces the flat file program traces with an indexed database that supports $O(1)$ lookup for the program trace cursor.

A number of different databases were considered for the role:

1. A custom database

If a custom database was written it could allow the abstract data types used to index the data to be specialised for the type of data being indexed. For example, n -ary trees might be more appropriate for steadily incrementing data such as the primary key than b -trees. B -trees might be more appropriate, on the other hand, for storing the values of memory locations and registers. The disadvantage of writing a custom database is the length of time required to get a correct implementation would probably be quite long.

2. The Berkeley Database[ber]

The Berkeley Database is a lightweight database system that supports transactions, joins and secondary indices. Since it is an embedded database there is no overhead from communicating with a separate server. The disadvantage of using the Berkeley Database is that it does not include a query processor so one would have to be written, although it is possible the SQL query processor in MySQL[mys] (which is based on the Berkeley Database) could be adapted.

3. SQLite[sql]

SQLite is an embedded SQL database engine. Like the Berkeley Database it has no overhead from communicating with a separate server. Unlike the Berkeley Database SQLite includes an SQL query processor that implements most of SQL92[sql92] so no separate query processor would have to be written. The disadvantages of using SQLite are that i) it is typeless and stores all data as strings so there is the extra overhead of converting integers to strings and ii) there is no low-level interface to write data to the database. SQLite 3, due for release shortly after the completion of this project, solves the first problem with manifest typing that includes support for native integers.

4. Other SQL databases

It is unlikely most other SQL databases would be suitable for the role as the overhead of communicating between the database client (the Logrind skin) and the server using IPC would be too great (although no tests were performed to confirm this).

Exploratory work was performed with both a custom database and a Berkeley DB based solution. However with both solutions difficulties were experienced i) writing a good query processor and ii) ensuring the correctness of the implementation, within the time constraints for the project. SQLite appeared to have the advantage over the other proposals when time was taken into consideration. SQLite is attractive because it already includes an SQL query processor and SQL is the standard database query language. Logrind 2 originally used an optimised version of SQLite as its database engine as discussed in section 5.2. The optimisations added support for integers stored in binary format as opposed to as strings, eliminating the overhead of string conversion. However this was sensitive to schema changes so we reverted to the vanilla implementation of the database. As mentioned above the latest version of SQLite due for release shortly after the deadline for this project has support for manifest typing including native integers. It should offer a performance increase (see section 6.1) exceeding the original optimisations which would have become superfluous.

SQLite supports pre-compiled queries and this feature is used in the Logrind2 skin to add the program trace data to the database. In addition the skin is also capable of writing directly to the B-tree structure used by SQLite by linking against some internal SQLite functions. Although this method of writing the data is faster it is sensitive to changes in data format of the rows in a table and the indices on the columns. For this reason the skin uses the slower pre-compiled queries by default.

Logrind 2 adds a number of commands to GDB for viewing and querying program traces. Each command has the `logrind` prefix to prevent name collisions with GDB's existing commands. The main command for viewing a portion of the program trace is the `logrind history` command. This command generates an SQL query from its arguments which is executed using SQLite.

When the command is invoked with the following syntax:

```
history START COUNT FILTER
```

this SQL query is executed:

```
select * from trace where FILTER and key > START limit COUNT;
```

A callback is called for each row which pretty prints the trace entry. In particular the callback will try to replace addresses with the corresponding symbol from the symbol table. Some symbols have a computed value. For example the addresses of local variables is a function of the stack pointer. In order to support these kinds of symbols the program trace cursor (see below) is temporarily set to the row being output. This means all register and memory reads made by GDB will return the value of the registers and memory at that point in the program trace. GDB can then calculate the address of each computed symbol. After the addresses are computed, the program trace cursor is restored to its previous value.

The program trace cursor can also be set manually using the `set logrind cursor` command. This allows Logrind 2 to offer similar functionality to Jonathon's code, i.e. it allows the user to view the memory and registers as

they would have been at a particular point in the program trace capture. In order to support this functionality Logrind 2 overrides the functions that read registers and memory from the inferior. If a program trace cursor is set these functions execute an SQL query specially formulated to return the memory or register contents at that point in the program trace (see section 4.5.4). If a program trace cursor is not set, the default functions are called instead.

The memory or register contents at a particular point in the trace may be found in the nearest pre-image of a memory or register write after the program cursor, post-image of a memory or register write before the cursor, or value of a memory or register read either side of the program cursor. See section 4.5.4 for more details.

4.5 The program trace query language

This section outlines the design of the program trace query language and discusses some its limitations. The language conforms to the SQL 92 [sql92] standard but has a fixed schema and defines extra functions.

4.5.1 Schema

The schema of the program trace database was designed with these goals in mind:

1. Make it easy to limit the size of the trace.
2. Minimise the number of joins required to evaluate a query.
3. Allow more than one run of a program to reside in the same database.
4. Support profiling.
5. Make it possible to uniquely specify a row.
6. Make queries execute quickly.

These goals each had a bearing on the final design. To support limits on the size of trace it is much easier to store all the trace events in a single table. That way the oldest row can be deleted to make room for a new one when the size limit is exceeded. Using a single table also limits the number of joins required to evaluate queries because everything is in the same table.

To allow more than one run of a program to reside in the same database the schema supports multiple sources where each source is a run of a program. The GDB commands can only work with one source at a time but the user can switch sources using the `set logrind source` command. This doesn't stop tools building on top of Logrind 2 making full use of the multiple program trace support however.

The schema includes an elapsed wall time column (`time`) to support profiling although it's worth noting that running a program under Logrind slows it down considerably (see section 5.2). A row may be generally be uniquely specified by this column but it is much more useful to have an auto incrementing integer perform the role as primary key.

Every column except the pre- and post- images is indexed. This is important because program traces can be very large and without indexing every query would take a long time to evaluate. Queries must be designed so that the dominating term is not a pre- or post- image column (see section 5.2).

The program trace schema also defines a number of views on the program trace — one view for each opcode. These views can make queries easier to understand and they prevent the user from accidentally referencing a column that's not applicable for a given opcode.

The design of the schema and the view definitions in particular owe a lot to Goldsmith's Program Trace Query Language [SGA04].

The full schema is given below:

```

create table Source (id integer primary key, sysname
    text, nodename text, release text, version text,
    machine text, pid integer, startTime integer);
create table Trace (id integer primary key, source
    integer, sequence integer, thread integer, time
    real, opcode integer, pc integer, data integer,
    preImage blob, postImage blob);
create index TraceSource on Trace (source);
create index TraceSequence on Trace (source, sequence);
create index TraceThread on Trace (source, thread);
create index TraceTime on Trace (source, time);
create index TraceOpcode on Trace (source, opcode);
create index TracePc on Trace (source, pc);
create index TraceData on Trace (source, data);
create view MemoryWrite as select id, source, sequence,
    thread, time, pc, data as address, preImage, postImage
    from Trace where opcode = trace_opcode_memwrite;
create view RegisterWrite as select id, source, sequence,
    thread, time, pc, data as register, preImage,
    postImage from Trace where opcode = trace_opcode_regwrite;
create view MemoryRead as select id, source, sequence,
    thread, time, pc, data as address, preImage as
    image from Trace where opcode = trace_opcode_memread;
create view RegisterRead as select id, source, sequence,
    thread, time, pc, data as register, preImage as
    image from Trace where opcode = trace_opcode_regread;
create view Jump as select id, source, sequence, thread,
    time, pc, data as address from Trace where opcode
    trace_opcode_jump;
create view Return as select id, source, sequence,
    thread, time, pc, data as address from Trace where
    opcode = trace_opcode_return;
create view SystemCall as select id, source, sequence,
    thread, time, pc, data as number, preImage as
    returnCode from Trace where opcode = trace_opcode_syscall;
create table dual (dummy text);
insert into dual values ('x');

```

The `dupl` table has the same role as in Oracle. If a user wishes to know the value of static function he or she may issue a `select` statement against the `dupl` table to ensure the result is only returned once.

Both the pre- and post- images of a memory or register write are stored in the database. The pre-image of a write is the value the location had before the write and the post- image is the value it has after the write (i.e. the value that was written). In a full program trace the pre- image of a write will be equal to the post- image of the last write to the same location (follows from program trace completeness — see appendix A). However in a partial program trace the value of the location may have been modified in the interim by some code that was not instrumented because the instrumentation was suppressed, therefore both the pre- and post- images are required

4.5.2 Functions

Logrind 2 defines a number of utility functions. Functions that vary according to scope take a `sequence` argument as a parameter that specifies where in the program trace to evaluate the function. Since some symbols are only valid in a particular scope Logrind 2 provides the `inscope` that returns true if a symbol is in scope.

eval(`sequence`, '`expr`')

Evaluates a source language expression at a particular point in the program trace.

Temporarily moves the program trace cursor then calls GDB's existing expression evaluation functions.

addr2line(`pc`)

Returns the source filename and line numbers corresponding to the given program counter address.

Uses GDB's existing symbol table functions.

line2addr('file:line')

Returns the start address of the given line.

Uses GDB's existing symbol table functions.

symbol2beginaddr(`sequence`, '`symbol`')

Returns the start address of the given symbol.

Uses GDB's existing symbol table functions.

symbol2endaddr(`sequence`, '`symbol`')

Returns the end address of the given symbol.

Uses GDB's existing symbol table functions.

addr2symbol(`sequence`, `address`)

Returns the name of the symbol at the given address plus some offset.

Uses a set of custom reverse symbol lookup functions.

inscope(`sequence`, '`symbol`')

Returns true if the given symbol is in scope else false.

Uses GDB's existing symbol table functions.

current_cursor()

Returns the current cursor position.

current_source()

Returns the id of the active source.

base64decode2int('data')

Converts the given base 64 data to an integer.

Uses Eric Raymond's `base64.c`.

base64decode2uint('data')

Converts the given base 64 data to an unsigned integer.

Uses Eric Raymond's `base64.c`.

hex2int('hex')

Converts the given hexadecimal number to an integer.

hex2uint('hex')

Converts the given hexadecimal number to an unsigned integer.

int2hex('hex')

Converts the given integer to a hexadecimal number.

uint2hex('hex')

Converts the given unsigned integer to a hexadecimal number.

4.5.3 Macros

Before a query is executed it is run through a macro processor which will re-write certain terms in the query. Logrind 2 provides built-in macros for the program trace opcodes (e.g. `opcode_memwrite = 1`) and register names (e.g. `register_ebx = 2`). In addition to the built-in macros a user may add his or her own macros using the `logrind macro define` command.

Logrind 2's macros use GDB's existing source macro support. This is designed to support evaluation of macros in the source code of a program being debugged but it proved to be easily extended to support macros in program trace queries.

4.5.4 Built-in queries

There are two interesting queries built into the Logrind GDB target (the remainder are trivial). The queries underly the program trace 'cursor' support. When a program trace cursor is set GDB's register and memory commands obtain their results from the program trace rather than the inferior process using these queries.

Query 4.1 retrieves the value a register had at the current cursor position. The query consists of two halves. The first half retrieves the pre-images of any reads or writes to a given register occurring after the program trace cursor. The second half retrieves the post-images of any writes to the register occurring before the cursor. To support partial program traces the closest matching row is used rather than the first before the cursor.

Query 4.2 retrieves the value of a memory region at the cursor position. This query consists of two halves similar to query 4.1, however unlike registers there may have been more than one write to a given memory region that conferred it its current value. The most recent write to each memory address in the region is required to reconstruct the value of the whole region. The results

Query 4.1: Query the value of a register at the cursor position

```
select
  image
from
  (select
    abs(sequence - current_cursor()) as
    distance,preImage as image
  from
    trace
  where
    (opcode = opcode_regread or (opcode
    = opcode_regwrite and sequence >=
    current_cursor())) and
    data = register number and
    source = current_source()
  union
  select
    abs(sequence - current_cursor() + 1) as
    distance,postImage as image
  from
    trace
  where
    opcode = opcode_regwrite and
    sequence < opcode_regwrite and
    data = register number and
    source = current_source()
  order by
    distance
  limit 1);
```

Query 4.2: Query the value of a memory region at the cursor position

```
select
  data,preImage
from
  trace
where
  (opcode = opcode_memread or (opcode
    = opcode_memwrite and sequence >=
    current_cursor())) and
  data >= start address and
  data < end address and
  source = current_source() and
  abs(sequence - current_cursor()) in
  (select
    min(abs(sequence - current_cursor()))
  from
    trace
  where
    opcode = opcode_memread or opcode =
    opcode_memwrite) and data >= start address
  and
  data < end address and
  source = current_source()
group by
  data)
union
select
  data,postImage
from
  trace
where
  opcode = opcode_memwrite and
  sequence < current_cursor() and
  data >= start address and
  data < end address and
  source = current_source() and
  abs(sequence - current_cursor()) in
  (select
    min(abs(sequence - current_cursor()))
  from
    trace
  where
    (opcode = opcode_memread or opcode =
    opcode_memwrite) and
    data >= start address and
    data < end address and
    source=current_source()
  group by
    data);
```

of this query are passed to a callback function that copies the image to a temporary buffer at the correct offset. The results are sorted by descending order of distance from the cursor so that in the case of overlapping memory writes the most recent value is used.

4.5.5 Results

The results of a query are presented in one of two ways depending on which command was used to execute it. The output of the `logrind history` command is a pretty printed program trace. An example trace is shown in figure 4.4.

If GDB is running interactively on a terminal different colours are used for the source lines, the assembly instructions and the trace lines to visually distinguish them.

The output of the `logrind query` command is shown in a table. An example is shown in figure 4.5. It would be useful to add an option to pretty print the output of this command as well, perhaps with any extra columns in the results shown as annotations on the trace. The current version of SQLite cannot store binary data in a column so the pre- and post- images are base64 encoded. This has the unfortunate side effect that the true values of the columns are obfuscated in query results so Logrind 2 provides a `base64decode` function to decode the data.

4.5.6 Limitations

The main limitation of the query language is inherited from SQL: sequences must be represented explicitly and require a join for each element in the sequence. A preferred alternative would be a regular expression language that ranged over the program trace, especially if it could be combined with the SQL query language.

Evaluating regular expressions on a relational database is not trivial however; it would take too long to evaluate the regular expression for every row in the trace. A more efficient algorithm is the following:

1. Generate the set of elements that must occur in a match (e.g. unquantified elements)
2. For each element in the set that is dominated by an indexed column, count how many occurrences of that element occur in the program trace. A boolean expression is dominated by a subexpression if the constraints given by the subexpression are stricter than any other subexpression. (e.g. the subexpression `A >= 3` dominates the expression `A >= 3 OR A = 3 OR TRUE`).
3. Select the element with the fewest occurrences as the anchor for the regular expression.
4. Evaluate the regular expression for each occurrence of the element in the trace using it as the anchor for the regular expression.

The author is of the opinion that a tool's functionality should not be artificially restricted and it is for this reason that Logrind 2 allows users to issue

Figure 4.4: An example of output from the `logrind history` command

```
progA.c:1 void _start (void) {
    push    %ebp
      74 (0.0164626): [1] _start (progA.c:1)
        regwrite esp: bffff550 => bffff54c
      75 (0.0165101): [1] _start (progA.c:1)
        memwrite bffff54c: bffff55c => 00000000
    mov     %esp,%ebp
      76 (0.0165574): [1] _start + 1 (progA.c:1)
        regwrite ebp: 00000000 => bffff54c
    sub     $0x8,%esp
      77 (0.0165927): [1] _start + 3 (progA.c:1)
        regwrite esp: bffff54c => bffff544
progA.c:2 int a = 1;
    movl   $0x1,0xffffffff(%ebp)
      78 (0.0166320): [1] _start + 6 (progA.c:2)
        memwrite a: _end + 939227037 => 00000001
progA.c:4 a = 2;
    movl   $0x2,0xffffffff(%ebp)
      79 (0.0166770): [1] _start + 19 (progA.c:4)
        jump _start + 19
      80 (0.0167171): [1] _start + 19 (progA.c:4)
        memwrite a: 00000001 => 00000002
progA.c:6 exit (0);
    sub     $0xc,%esp
      81 (0.0167591): [1] _start + 26 (progA.c:6)
        regwrite esp: bffff544 => bffff538
    push   $0x0
      82 (0.0167927): [1] _start + 29 (progA.c:6)
        regwrite esp: bffff538 => bffff534
      83 (0.0168833): [1] _start + 29 (progA.c:6)
        memwrite bffff534: 00000001 => 00000000
    call   0x8048194
      84 (0.0169495): [1] _start + 31 (progA.c:6)
        regwrite esp: bffff534 => bffff530
      85 (0.0169896): [1] _start + 31 (progA.c:6)
        memwrite bffff530: bffff554 => _start + 36
      86 (0.0170452): [1] _start + 31 (progA.c:6)
        call _end + 133611988
```

Figure 4.5: An example of output from the `logrind query` command

```
id source sequence thread time opcode pc data preImage postImage
74 1 74 1 0.0164626 2 134513060 8 UPX/vw== TPX/vw==
75 1 75 1 0.0165101 1 134513060 3221222732 XPX/vw== AAAAAA==
76 1 76 1 0.0165574 2 134513061 7 AAAAAA== TPX/vw==
77 1 77 1 0.0165927 2 134513063 8 TPX/vw== RPX/vw==
78 1 78 1 0.0166320 1 134513066 3221222728 FQoAQA== AQAAAA==
79 1 79 1 0.0166770 5 134513079 134513079 null null
80 1 80 1 0.0167171 1 134513079 3221222728 AQAAAA== AgAAAA==
81 1 81 1 0.0167591 2 134513086 8 RPX/vw== OPX/vw==
82 1 82 1 0.0167927 2 134513089 8 OPX/vw== NPX/vw==
83 1 83 1 0.0168833 1 134513089 3221222708 AQAAAA== AAAAAA==
84 1 84 1 0.0169495 2 134513091 8 NPX/vw== MPX/vw==
85 1 85 1 0.0169896 1 134513091 3221222704 VPX/vw== yIEECA==
86 1 86 1 0.0170452 6 134513091 134513044 null null
13 rows
```

queries against the whole of the program trace. However a user may issue such a query by accident so we added a mechanism for interrupting long-running queries. After a query has been executing for 200ms a prompt will be shown `Press Ctrl-C to interrupt query...` The user may follow the instructions to stop the query. It would be useful if the tool could suggest a way of re-writing a query to take advantage of indices if possible, although this feature is non-trivial and was not implemented

4.6 Summary

There have been a number of important design decisions for Logrind 2 such as the communications mechanism between Valgrind and GDB; the choice of database to store the program trace; and the program trace schema. The choice of shared memory for communicating between Valgrind and GDB was influenced by Reiss's argument but is, in hindsight, not forward compatible with future versions of Valgrind. Nevertheless on the whole the decisions taken have made possible a robust implementation of the original specification.

Chapter 5

Evaluation

This chapter: summarises the testing procedure for Logrind 2 and the changes that were made as a result; gauges the overhead due to program trace capture; examines the efficiency of the program trace query language; comments on the documentation that was written for the tool; and finally looks at how suitable Logrind 2 is for an open source release.

5.1 Testing

The implementation of Logrind 2 was tested thoroughly to ensure it met the specification. Wherever possible automated tests were used although some tests by their very nature required manual inspection. The design of the tests is somewhat arbitrary but they focus on i) the most complex functions and ii) parts of the code that have been problematic in the past. The tests were designed as unit tests but because of the integration between the components in Logrind 2 they generally exercise more than one component at once. As a result of performing the tests bugs were discovered in the implementation which were duly fixed and the tests re-run. This may be taken as evidence that such testing is a necessary part of software development. The full list of tests that were performed and their results can be found in appendix B.

Test 4 (Maximum database size option is honoured) revealed that the Logrind 2 skin was not respecting the file size limit set using the `set logrind max-database-size` command. Unfortunately the cause of the problem was not diagnosed before the project deadline but this should not affect the remaining functionality of the tool.

Test 7 ("logrind history" command accepts the argument format: -START) failed because the whole trace was printed rather than the last 10 lines. The problem was identified as two source code lines in the wrong order and fixed.

Test 10 (Program trace correctly reflects the operations performed by the program) failed. A cursory examination of the program trace output revealed function calls, jumps and returns were not being traced by the Logrind skin. The cause of the problem was identified as a missing `break` statement and fixed.

Test 16 (Program trace cursor can be switched on/off) revealed a flaw in query 4.2 that is used to retrieve the value of a memory region. If the cursor

was pointing directly at a memory write the post-image of the write was used instead of the pre-image. The query was corrected to fix the problem.

The results of the tests show that Logrind 2 broadly meets its original specifications and provides a robust implementation of them.

5.2 Program trace capture overhead

One of the unfortunate problems with program tracing in software is that a comparatively large number of instructions needs to be executed for every traced instruction in the original program. This slows the execution of the program down considerably. It would be useful to quantify this overhead to assess its significance.

The well known BYTEmark benchmark program [byt] was used to generate a metric for comparing normal execution of a program; execution of a program under Logrind 2 but with no instrumentation; and execution of a program under the Logrind 2 skin with capture of different sets of events turned on. This reflects that applications that use program traces differ in the information they require. A profiling tool might only be interested in function entry and exit points, for example, and not memory writes.

The results of the benchmarks are shown in table 5.1 and figure 5.1.

There is an obvious overhead from just running a program under the Logrind 2 skin even if no instrumentation is added. This overhead is partly due to the simulation process employed by Valgrind that translates each block of straight line code the first time it is encountered. However the Valgrind documentation suggests this slowdown is only 4 times so a slowdown of 60 times revealed by the benchmark was surprising. Normal Valgrind skins generate an error when some constraint is broken. Valgrind's suppression mechanism can be used to suppress these errors. Since errors are rare the overhead of the suppression lookup mechanism is not significant. Logrind 2, on the other hand, evaluates suppression at instrumentation time — the suppressions control what instrumentation is added. The overhead of suppression lookup is therefore higher for Logrind 2 because it is performed for every basic block. However the 60 times slowdown was unexpectedly high. Close examination of the code revealed a design flaw: suppression lookup was being performed before the translation of every instruction. The code was modified to cache the results of suppression lookup for a particular event type. Unfortunately this did not give any significant speed improvement, probably because basic blocks are usually small so there are few cache hits. There was not time to re-run all the benchmarks after the code fix so the remainder of this discussion will use the old results.

When instrumentation is turned on there is an even larger overhead because a database insert occurs for every instrumented instruction. There is little that can be done to reduce this overhead without a corresponding trade-off in query performance. A flat program trace with no indices would be fast to write but any queries against it would have to read every row — this would be very slow! One optimisation is to write directly to the database's B-tree structure rather than relying on SQLite's SQL virtual machine. However such an implementation is fragile and sensitive to changes in the program trace schema. An early version of the Logrind 2 implementation successfully used this ap-

proach but it was abandoned in favour of using pre-compiled SQL for the sake of clarity and reliability.

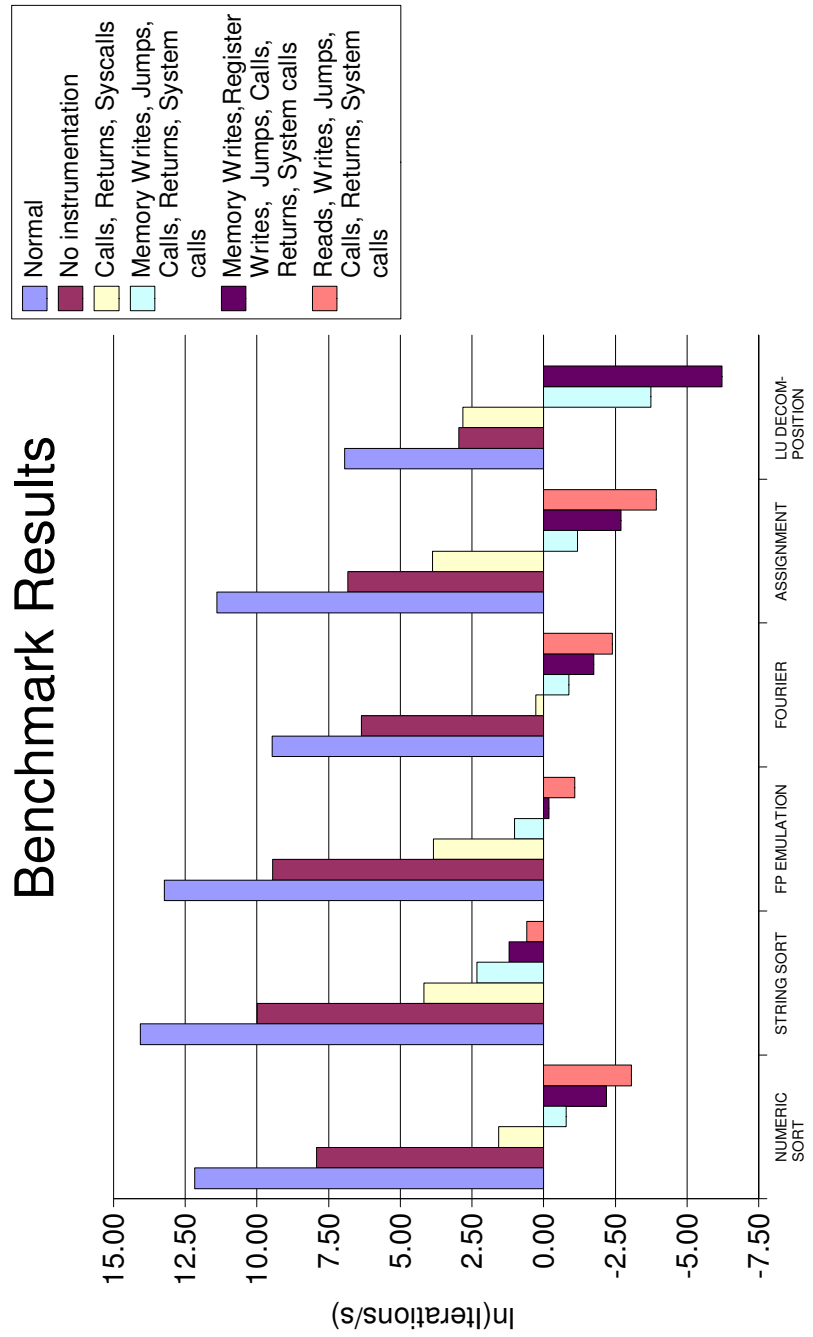
Although the overhead of program tracing using Logrind 2 might seem prohibitively high, in reality debugging usually focuses on one part of a project at once. A developer working on code in Mozilla for submitting forms to the web will probably not be interested in the code that controls user interface layout. As such a developer can use Logrind 2's suppression support to limit program trace capture to the part of the program he or she is working on. In this way the slowdown may be reduced to an acceptable level.

Table 5.1: Benchmark results using BYTEmark Native Mode Benchmark

Test	Normal execution		No instrumentation		Calls, returns and system calls		Memory writes, jumps, calls, returns and system calls		Register, writers, Memory writes, jumps, calls, returns and system calls		Reads, writes, jumps, calls, returns and system calls	
	Iterations/s	Index	Iterations/s	Index	Iterations/s	Index	Iterations/s	Index	Iterations/s	Index	Iterations/s	Index
Numeric sort	192300	1.000	2731	1.420×10^{-2}	4.796	2.494×10^{-5}	0.4615	2.400×10^{-6}	0.1128	5.869×10^{-7}	4.740 $\times 10^{-2}$	2.465×10^{-7}
String sort	1273000	1.000	31670	1.702×10^{-2}	64.96	5.105×10^{-5}	10.302	8.095×10^{-6}	3.328	2.615×10^{-6}	1.801	1.415×10^{-6}
Emulated floating-point	554100	1.000	12720	2.296×10^{-2}	46.79	8.443×10^{-5}	2.783	5.022×10^{-6}	0.8300	1.498×10^{-6}	0.339	6.118×10^{-7}
Fourier coefficients	12980	1.000	572.2	4.407×10^{-2}	1.312	1.010×10^{-4}	0.4144	3.191×10^{-5}	0.1742	1.341×10^{-5}	9.205 $\times 10^{-2}$	7.087×10^{-6}
Assignment	89420	1.000	930.8	1.041×10^{-2}	48.32	5.404×10^{-4}	0.3067	3.430×10^{-6}	6.789 $\times 10^{-2}$	7.593×10^{-7}	1.970 $\times 10^{-7}$	2.203×10^{-7}
LU decomposition	1042	1.000	19.36	1.857×10^{-2}	16.68	1.601×10^{-2}	2.378 $\times 10^{-2}$	2.262×10^{-5}	2.004 $\times 10^{-3}$	1.923×10^{-6}	1	2.334×10^{-7}
Mean index	1.000	1.000	1.639 $\times 10^{-2}$	61.02	8.016 $\times 10^{-3}$	124.8	79300	79300	79300	4285000		

¹ This result is missing because the program trace database filled the hard disk on the test machine before completion.

Figure 5.1: Benchmark results using BYTEmark Native Mode Benchmark



5.3 Query efficiency

It is important that queries on the program trace work very efficiently because the trace may be long for even simple programs. Generally the indices in Logrind 2's program trace schema ensure queries are efficient but certain queries involving functions may need to be re-written to take full use of them.

The most frequent queries are the two built into the Logrind GDB target used to retrieve register (query 4.1) and memory values (query 4.2).

All of the columns referenced in the `where` clause of query 4.1 are indexed. Even so this query reads every row that contains a read or write to the given register because this is required to sort them as specified by the `order by` clause. A more efficient version of the query might avoid the sort by scanning the table sequentially either side of the cursor position. Unfortunately SQL makes no guarantees about the order of rows in a table so a query in this form would be making assumptions about the layout of the table. Furthermore in some traces it might be necessary to scan many rows before finding a match. For these reasons the optimised query was not used in Logrind 2.

Query 4.2 suffers from the same problem as the previous query, namely that the `min` aggregate function requires every memory read or write that access the given region to be read from the table. Memory writes may be spread sparsely in the table so sequentially scanning the table either side of the cursor position would probably not be very efficient either. Although there may be a more efficient way of performing this query we have not found one.

Certain queries are particularly inefficient, namely those containing functions of columns in the trace table.

Consider for example:

```
select * from trace where addr2line(pc)="abc:c:6"
```

To evaluate this query the database engine must read every row from the trace table — and this may take some time! However Logrind 2 anticipates this problem and provides reverse counterparts to every function. For example the reverse function for `addr2line` is `line2addr`. The above query can be rewritten as:

```
select * from trace where line2addr("abc:c:6")=pc1
```

The function can be evaluated statically since it does not rely on the value of any columns. This static value may then be looked up in the `TracePc` index to retrieve the matching rows. This query is more efficient than the previous one.

5.4 Documentation

As per the original specification comments were added to the source code and a unified user guide written to cover the Logrind 2 skin and the GDB target. Unfortunately because of time constraints it was not possible to write developer documentation for all of the components although the existing documentation was updated to cover the interactive debugging support for Valgrind.

¹This is not exactly equivalent to the previous query because a line may span more than one program counter address. However the Dwarf 2 debugging information format [Int93] does not include information about the highest address corresponding to a line — this must be inferred and is prone to error.

The user guide could benefit from the addition of some examples and an introduction to program trace technology but this was omitted, again because of time constraints.

5.5 Open source

The interactive debugging support for Valgrind was implemented before the other deliverables. It included user documentation and useful comments in the source code. This code was released as two patches and made available for download on a website. A post was sent to the Valgrind developers' mailing list advertising the new functionality. In summary the responses made three important points: i) they advised against doing any major work against version 2.0.0 because the latest development version was incompatible ii) it would be nice not to expose Valgrind internal structures iii) using a GDB stub might be a better solution. While it was clear that the new functionality was welcome the developers were reluctant to integrate the patches because i) they were against an old version of Valgrind and ii) they didn't believe that using shared memory was the 'right way' to add debugging support. Due to time constraints it was not possible to rewrite Logrind 2 so these issues were not present.

Clearly getting patches integrated with the main release of an open source or free software project requires more than just good code or a good idea. Open source developers usually offer their time for free and can't afford to work on other peoples' code to make it suitable for merging with the CVS head, for example. Furthermore if a particular patch doesn't fit with the developers' ideology it is unlikely to be accepted. The GDB developers, for example, would probably not accept a large patch that added the program trace functionality from Logrind 2 because it would be heavily tied to Valgrind and the Logrind skin. Contributing to the free software community requires skill in politics as well as programming.

In summary more work focussed on integration is required for Logrind 2 to be integrated with the main releases of Valgrind or GDB. Our experience has lead us to conclude the following criteria are important to a large patch being accepted by an open source project:

1. The patch must be against the latest development sources (often the CVS head).
2. The patch should rely on other projects and tools as little as possible.
3. The patch must follow the existing design philosophies of the project.

5.6 What people are saying about Logrind 2

About interactive debugging support for Valgrind:

"This sounds quite cool" — Nicholas Nethercote, University of Cambridge

"Sounds quite cool." — Josef Weidendorfer, Institut für Informatik der TU München

"That's pretty cool." — Jeremy Fitzhardinge

"Your Valgrind target for GDB looks great. It's probably what I'm looking for

— I'm using GDB to produce trace data that is then given to gnuplot in order to get graphs. My problem is that GDB does introduce an enormous overhead on the running program, and therefore I can't rely on any timing information for the data.

Using the Valgrind target, I'm hoping to be able to get some information about, say, CPU cycles spent on the program, to get a reliable measure. Can you enlighten me on whether/how this is possible?

Also, I'd strongly advise that you ask for your patches to be integrated into official branches. I don't think they'd break anything, and they do what lots of people have been waiting for: connecting GDB and Valgrind.

This is some fine work!" — 'Gnurou'

About the whole project:

"I'm interested indeed. I've already thought that using a gdb/valgrind combo it should be possible to precisely measure the executions times in number of cycles. Is it what you are doing? Do you have informations about this and whether it is possible?

Please keep me informed if possible about your project once it becomes usable!" — Alexandre Courbot

5.7 Summary

Testing is an important part of the software engineering process. Testing Logrind 2 allowed us to uncover bugs that were subsequently fixed and demonstrated that the project broadly meet the original specifications.

Although there is a significant overhead caused by program trace capture in software we believe that this does not make Logrind 2 unsuitable for real world applications because it provides features that allow a user to specify exactly which parts of a program he or she wishes to trace. There is scope for optimising the capture storage.

The program trace schema used by Logrind 2 allows the most common program trace queries to take advantage of indices on the trace table. Unfortunately the use of ordering in the queries means some sequential scanning is still required. It may be possible to redesign the schema to be more efficient.

Although people have commented favourably about the project further work is required before Logrind 2 can be integrated with the main releases of Valgrind or GDB.

Chapter 6

Future work

Logrind 2 is designed to be a program trace framework that other tools can use. As such the scope of future work is very wide. The discussion here will focus on work relating to the framework itself rather than tools that could use it.

6.1 Optimisation

It is possible to optimise the recording of the program trace. SQLite 3 (due for release July 1 2004) introduces manifest typing that supports binary integers and blobs. Using SQLite 3 would reduce the overhead of converting to strings before inserting it into the database. As mentioned earlier the overhead of SQLite's virtual machine could be eliminated by writing directly to the B-tree structure on disk using SQLite's internal routines.

It may also be possible to optimise the program trace schema as mentioned in section 5.3. Careful choice of queries for the register and memory value lookups could improve the speed of the `logrind history` command substantially.

6.2 Integration with the main releases of Valgrind and GDB

This will probably require the interactive debugging support to be re-written to use the GDB stub mechanism. The Valgrind skin and core code will need to be updated to the most recent CVS release as will GDB. Most of the work achieving the integration, however, is in politics: convincing the respective developers that Logrind 2 will be a useful addition to their codebase.

6.3 Better support for multiple sources

The Logrind 2 GDB target allows only one source to be active at a time (although multiple sources may exist in the database). To support delta debugging and other such techniques it would be useful to add commands to support

multiple sources simultaneously. Similarly new versions of the built-in functions such as `inscope` could be added that take an extra source parameter.

6.4 Program trace annotations

Only the results of the `logrind history` command are pretty-printed. It would be good to pretty-print the output of `logrind query` commands where appropriate, annotating each line with any extra columns that occur in the result.

6.5 Graphical user interface

A graphical user interface could improve the user experience of Logrind 2 by providing better visualisation of the program trace. For example dynamic program slices output by the `logrind history` command could be shown on Eclipse's [\[ecl\]](#) program Visualiser widget found in the AJDT and CME subprojects.

6.6 Drill-down on program history

The `logrind history` command offers no way to drill-down into a program trace although this can be clumsily emulated using filters. The addition of a graphical user interface would make it easy to offer a drill-down feature using a virtual tree widget.

6.7 Dependence capture

Although Logrind 2 traces both reads and writes it does not record the relationship between them. For example an add operation will show as two reads and a write but there is no mechanism for determining that the value of the write depends on the value of the preceding reads. In order to support dependence analysis Logrind 2 will need to track the origin of values as they are manipulated. The source operand(s) of an instruction may be one of Valgrind's temporary register so these will need to be tracked as well. Nethercote and Mycroft discuss a method for performing dataflow analysis using Valgrind in their paper "Redux: A Dynamic Dataflow Tracer" [\[NM03\]](#).

6.8 Macro library

Useful queries could be collected in a macro library distributed with the project to reduce the burden on the user writing their own queries.

6.9 Improved documentation

The developer documentation for Valgrind and GDB needs to be finished to include information about the Logrind 2 Valgrind skin and the Logrind 2 GDB target. The user guide could also benefit from an introduction and some examples.

Chapter 7

Conclusions

The decisions taken in the design of Logrind 2 have made possible a robust implementation of the project. We feel the final overall design is a good solution to the original specifications. The importance of testing cannot be overestimated and the test suite made a valuable contribution to the development process. A greater number of tests with wider coverage would have been even more beneficial had there been time. Time constraints imposed by the project deadline made prioritising goals very important. This inevitably meant that some lower priority items, such as the developer documentation, were incomplete. Nevertheless we feel the priorities chosen were appropriate with no major functionality sacrificed.

The benchmark results from testing the Logrind 2 skin reveal the inevitable overhead due to program trace capture but these are not unreasonable given the processing required. We believe that through judicious use of suppressions the overhead may be reduced to a manageable level. Furthermore the reduced execution speed could be considered an acceptable tradeoff for increased productivity.

This project makes program trace technology accessible to the average developer. Developers who are already familiar with the well known Valgrind and GDB tools will immediately benefit from the integrated program trace support. The implemented functionality is sufficient for the debugging tasks listed in the introduction (broken constraints, unexpected control flow and data corruption) as well as historical state inspection, query-based debugging and time profiling.

Logrind 2 provides a framework upon which many program trace tools may be built. The decision to use a standard SQL based database to store the program traces will make it easy for developers to write their own tools. Furthermore these tools are not restricted to being integrated with GDB: they may be stand-alone or built into integrated development environments such as Eclipse. Logrind 2 removes key barriers to delivering program trace based debugging on real applications. There has been interest from the free software community in Logrind 2 and future work may lead to integration with Valgrind and/or GDB. This would make it even easier for developers to extend the project.

Bibliography

- [ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616, 1993. 14, 18
- [ber] Berkeley database. <http://www.sleepycat.com/>. 31
- [byt] Bytemark. <http://www.byte.com/bmark/bdoc.htm>. 43
- [CFC01] Shyh-Kwei Chen, W. Kent Fuchs, and Jen-Yao Chung. Reversible debugging using program instrumentation. *IEEE Trans. Softw. Eng.*, 27(8):715–727, 2001. 14, 18
- [CL96] Christopher Colby and Peter Lee. Trace-based program analysis. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 195–207. ACM Press, 1996. 13, 57
- [Cod83] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, 1983. 19
- [Coo03] Jonathon Cooper. Detailed program tracing and replaying: Theory and implementation. Master’s thesis, Imperial College, London, 2003. 6, 16
- [Dai] Daikon. <http://pag.csail.mit.edu/daikon/>. 19
- [ddd] Data display debugger. <http://www.gnu.org/software/ddd>. 13
- [ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE ’99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999. 6, 15
- [ecl] Eclipse. <http://www.eclipse.org>. 51
- [FH93] Peter H. Feiler and Watts Humphrey. Software process development and enactment: Concepts and definitions. In *Proceedings of Second International Conference on the Software Process*. IEEE Computer Society Press, February 1993. 13

- [gdb] The gnu project debugger. <http://www.gdb.org>. 6
- [gnu] Gnu's not unix. <http://www.gnu.org>. 13
- [gpl] Gnu general public license. <http://http://www.gnu.org/copyleft/gpl.html>. 12
- [Hsi93] P. Hsin. Software debugging with dynamic instrumentation and test-based knowledge, 1993. 15
- [ins] Insight - the gdb gui. <http://sources.redhat.com/insight>. 13
- [Int93] UNIX International. *DWARF Debugging Information Format*, 1993. 47
- [Jin] Jinsight. <http://www.research.ibm.com/jinsight/docs/faqs.htm>. 17
- [KHH⁺01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001. 19
- [Lew03] B. Lewis. Debugging backwards in time. In K. De Bosschere M. Ronsse, editor, *Processings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, Ghent, 2003. 16
- [LHS97] Raimondas Lencevicius, Urs Hoelzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 304–317. ACM Press, 1997. 15
- [LHS99] Raimondas Lencevicius, Urs Hoelzle, and Ambuj K. Singh. Dynamic query-based debugging. Technical report, 1999. 15
- [mys] Mysql. <http://www.mysql.com/>. 31
- [NM03] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003. 18, 51
- [NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999. 20
- [ODB] The omniscient debugger. <http://www.lamdacs.com/debugger/debugger.html>. 16
- [Rei93] Steven P. Reiss. Trace-based debugging. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 305–314. Springer-Verlag, 1993. 17, 19, 27, 28
- [Ret] Retrovue. <http://www.visicomp.com/>. 16

- [RR99] Manos Renieris and Steven P. Reiss. Almost: exploring program traces. In *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM international conference on Information and knowledge management*, pages 70–77. ACM Press, 1999. 18, 19, 23
- [SGA04] Robert O’Callahan Simon Goldsmith and Alex Aiken. Lightweight instrumentation from relational queries over program traces. Technical Report UCB/CSD-4-1315, Computer Science Division (EECS), University of California, Berkley, March 2004. 14, 19, 34
- [Slo97] Anthony M. Sloane. Generating dynamic program analysis tools. In *the Australian Software Engineering Conference (ASWEC’97)*, pages 166–173, Sydney, 1997. IEEE CS Press. 14
- [sql] Sqlite - an embeddable database engine. <http://www.sqlite.org/>. 31
- [sql92] *Database Language SQL*, 1992. ISO/IEC 9075. 19, 31, 33
- [Tip94] Frank Tip. A survey of program slicing techniques. Technical report, 1994. 21
- [vg] Valgrind. <http://valgrind.kde.org>. 6
- [Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981. 20
- [Zel73] M. V. Zelkowitz. Reversible execution. *Commun. ACM*, 16(9):566, 1973. 14
- [Zel99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999. 8

Website URLs are valid 25 May 2004. Historical websites can usually be found on the Internet Archive (<http://www.archive.org>).

Appendix A

Semantics of program traces generated by Logrind

We can define the operational semantics of the program traces generated by Logrind in a similar way to programming languages and use them to show that the information captured by Logrind in a program trace is sufficient to reconstruct the state of an x86 Linux executable.

Note that the semantics below borrow some concepts from transition system semantics [CL96]. The events in a Logrind program trace may be modelled as transfer functions since they only partially define the new state of the process.

Let us define:

$$val ::= \{byte \cup word \cup dword \cup \dots\}$$
$$addr ::= dword$$
$$reg ::= \{eip, eax, ebx, \dots\}$$
$$state ::= (addr \cup reg) \times val$$
$$log ::= \left(\begin{array}{l} \langle log_opcode_regwrite, addr, reg, val \rangle^* \vee \\ \langle log_opcode_memwrite, addr, addr, val \rangle^* \vee \\ \langle log_opcode_jump, addr, addr \rangle \end{array} \right)^*$$
$$instr ::= \{\dots\}$$
$$s \in state$$
$$l \in log$$

a translation relation that models the processor executing a machine code instruction:

$$\langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

and a function:

$$S : state \times val$$

that gives the size of the instruction at the program counter *eip*.

We may partially define some rules for \rightsquigarrow_C :

$$\langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

execute_writes

1 ...

$$2 \exists x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s'(x) \neq s(x)$$

$$3 s'(\text{eip}) = s(\text{eip}) + S(s)$$

$$\langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

execute_jump

1 ...

$$2 \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s'(x) = s(x)$$

$$3 s'(\text{eip}) \neq s(\text{eip}) + S(s)$$

$$\langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

execute_other

1 ...

$$2 \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s'(x) = s(x)$$

$$3 s'(\text{eip}) = s(\text{eip}) + S(s)$$

$$\langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

We may define another translation relation \rightsquigarrow_L that models the logging behaviour of Logrind. This relation has one rule that models Logrind recording memory write and register write events, another logging jumps and a third rule that models any other instruction.

\rightsquigarrow_L has the signature:

$$\langle s, l \rangle \rightsquigarrow_C \langle s', l' \rangle$$

and the rules:

log_writes

$$1 \langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

$$2 l = \left\{ \begin{array}{l} j \\ j = \langle \text{log_opcode_memwrite}, s(\text{eip}), x, s'(x) \rangle \wedge x \in \text{addr} \wedge s(x) \neq s'(x) \cup \\ j = \langle \text{log_opcode_regwrite}, s(\text{eip}), x, s'(x) \rangle \wedge x \in (\text{reg} - \text{eip}) \wedge s(x) \neq s'(x) \end{array} \right\}$$

$$3 \exists x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \wedge s'(x) \neq s(x)$$

$$4 s'(\text{eip}) = s(\text{eip}) + S(s)$$

$$\langle s, l \rangle \rightsquigarrow_L \langle s', l.l \rangle$$

log_jump

$$1 \langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

$$2 \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s'(x)$$

$$3 s'(\text{eip}) \neq s(\text{eip}) + S(s)$$

$$4 l = \langle \text{log_opcode_jump}, s(\text{eip}), s'(\text{eip}) \rangle$$

$$\langle s, l \rangle \rightsquigarrow_L \langle s', l.l \rangle$$

log_other

$$1 \langle s \rangle \rightsquigarrow_C \langle s' \rangle$$

$$2 \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s'(x)$$

$$3 s'(\text{eip}) = s(\text{eip}) + S(s)$$

$$\langle s, l \rangle \rightsquigarrow_L \langle s', l \rangle$$

Now we may define a third translation relation \rightsquigarrow_P that models the Logrind target vector for GDB playing back a program trace.

\rightsquigarrow_P has the same signature as \rightsquigarrow_L .

$$\begin{array}{c}
\text{playback}_{writes} \\
1 = \left\{ j \mid \begin{array}{l} j = \langle \text{log_opcode_memwrite}, s(eip), x, s'(x) \rangle \wedge x \in \text{addr} \wedge s(x) \neq s'(x) \cup \\ j = \langle \text{log_opcode_regwrite}, s(eip), x, s'(x) \rangle \wedge x \in (\text{reg-eip}) \wedge s(x) \neq s'(x) \end{array} \right\} \\
2s(eip) = s(eip) + S(s) \\
\hline
\langle s, ls \rangle \rightsquigarrow_L \langle s', l.ls \rangle \\
\text{playback}_{jump} \\
1\forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s'(x) \\
2s'(eip)! = s(eip) + S(s) \\
3l = \langle \text{log_opcode_jump}, s(eip), s'(eip) \rangle \\
\hline
\langle s, ls \rangle \rightsquigarrow_L \langle s', l.ls \rangle \\
\text{playback}_{other} \\
1\forall x : x \in \text{addr} \cup (\text{reg-eip}) \rightarrow s(x) = s'(x) \\
2s'(eip) = s(eip) + S(s) \\
\hline
\langle s, ls \rangle \rightsquigarrow_L \langle s', ls \rangle
\end{array}$$

We may note that the semantics of \rightsquigarrow_P are almost identical to the semantics of \rightsquigarrow_L .

We want to show that replaying the program trace generated by Logrind is equivalent to the original execution of the program. That is to say:

$$\langle s, l \rangle \rightsquigarrow_P \langle s', l' \rangle \wedge \langle s, l \rangle \rightsquigarrow_L \langle s'', l' \rangle \rightarrow s' = s''$$

Note that recreating the final state of a process using its Logrind program trace requires us to start with the same initial state. Fortunately the initial state of a Linux process is the same each time it is run for a given Linux kernel version.

Proof:

Assume $\langle s, ls \rangle \rightsquigarrow_P \langle s', l.ls \rangle \wedge \langle s, l \rangle \rightsquigarrow_L \langle s'', l.ls \rangle$:

$$\begin{array}{l}
s'(eip) = s(eip) + S(s) \wedge \exists x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \wedge s'(x) \neq s(x) \vee \\
s'(eip) \neq s(eip) + S(s) \wedge \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s'(x) \vee \\
s'(eip) = s(eip) + S(s) \wedge \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s'(x) \\
s''(eip) = s(eip) + S(s) \wedge \exists x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \wedge s''(x) \neq s(x) \vee \\
s''(eip) \neq s(eip) + S(s) \wedge \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s''(x) \vee \\
s''(eip) = s(eip) + S(s) \wedge \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s''(x)
\end{array}$$

Assume $s'(eip) = s(eip) + S(s) \wedge \exists x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \wedge s'(x) \neq s(x)$:

Assume $s''(eip) = s(eip) + S(s) \wedge \exists y : y \in \text{addr} \cup (\text{reg} - \text{eip}) \wedge s''(y) \neq s(y)$:

Assume $\exists z : s'(z) \neq s''(z)$:

Assume $z = eip$:

$$s''(eip) = s(eip) + S(s)$$

$$s'(eip) = s(eip) + S(s)$$

$$s'(eip) = s''(eip)$$

\perp

Assume $z \in \text{addr}$:

$$\text{Assume } s'(z) = s(z) \wedge s''(z) \neq s(z) :$$

$$\begin{array}{l}
\neg\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(\text{eip}), z, s''(z) \rangle \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(\text{eip}), z, s''(z) \rangle \\
\perp \\
\text{Assume } s'(z) \neq s(z) \wedge s''(z) = s(z) : \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(\text{eip}), z, s'(z) \rangle \\
\neg\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(\text{eip}), z, s'(z) \rangle \\
\perp \\
\perp \\
\text{Assume } z \in (\text{reg} - \text{eip}) : \\
\text{Assume } s'(z) = s(z) \wedge s''(z) \neq s(z) : \\
\neg\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(\text{eip}), z, s''(z) \rangle \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(\text{eip}), z, s''(z) \rangle \\
\perp \\
\text{Assume } s'(z) \neq s(z) \wedge s''(z) = s(z) : \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(\text{eip}), z, s'(z) \rangle \\
\neg\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(\text{eip}), z, s'(z) \rangle \\
\perp \\
\perp \\
\perp \\
\neg\exists z : s'(z) \neq s''(z) \\
s' = s'' \\
\text{Assume } s''(\text{eip}) \neq s(\text{eip}) + S(s) \wedge \forall y : y \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(y) = s''(y) : \\
\text{Assume } x \in \text{addr} : \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(\text{eip}), x, s'(x) \rangle \\
l = \langle \text{log_opcode_jump}, s(\text{eip}), s''(\text{eip}) \rangle \\
\perp \\
\text{Assume } x \in (\text{reg} - \text{eip}) : \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(\text{eip}), x, s'(x) \rangle \\
l = \langle \text{log_opcode_jump}, s(\text{eip}), s''(\text{eip}) \rangle \\
\perp \\
\perp \\
s' = s'' \\
\text{Assume } s''(\text{eip}) = s(\text{eip}) + S(s) \wedge \forall y : y \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(y) = s''(y) : \\
\text{Assume } x \in \text{addr} : \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(\text{eip}), x, s'(x) \rangle \\
l = \emptyset \\
\perp \\
\text{Assume } x \in (\text{reg} - \text{eip}) : \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(\text{eip}), x, s'(x) \rangle \\
l = \emptyset \\
\perp \\
\perp \\
s' = s'' \\
s' = s'' \\
\text{Assume } s'(\text{eip}) \neq s(\text{eip}) + S(s) \wedge \forall x : x \in \text{addr} \cup (\text{reg} - \text{eip}) \rightarrow s(x) = s'(x) : \\
\text{Assume } s''(\text{eip}) = s(\text{eip}) + S(s) \wedge \exists y \in \text{addr} \cup (\text{reg} - \text{eip}) \wedge s''(y) \neq s(y) : \\
\text{Assume } y \in \text{addr} : \\
l = \langle \text{log_opcode_jump}, s(\text{eip}), s'(\text{eip}) \rangle \\
\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(\text{eip}), x, s''(x) \rangle \\
\perp
\end{array}$$

Assume $y \in (reg - eip)$:
 $l = \langle \text{log_opcode_jump}, s(eip), s'(eip) \rangle$
 $\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(eip), x, s''(x) \rangle$
 \perp
 \perp
 $s' = s''$
 Assume $s''(eip) \neq s(eip) + S(s) \wedge \forall y : y \in addr \cup (reg - eip) \rightarrow s(y) = s''(y)$
 Assume $\exists z : s'(z) \neq s''(z)$:
 Assume $z = eip$:
 $l = \langle \text{log_opcode_jump}, s(eip), s'(eip) \rangle$
 $l = \langle \text{log_opcode_jump}, s(eip), s''(eip) \rangle$
 $s'(eip) = s''(eip)$
 \perp
 Assume $z \in addr \cup (reg - eip)$
 $s'(z) = s(z)$
 $s''(z) = s(z)$
 $s'(z) = s''(z)$
 \perp
 \perp
 $\neg \exists z : s'(z) \neq s''(z)$
 $s' = s''$
 Assume $s''(eip) = s(eip) + S(s) \wedge \forall y : y \in addr \cup (reg - eip) \rightarrow s(y) = s''(y)$:
 $l = \langle \text{log_opcode_jump}, s(eip), s'(eip) \rangle$
 $l = \emptyset$
 \perp
 $s' = s''$
 $s' = s''$
 Assume $s'(eip) = s(eip) + S(s) \wedge \forall x : x \in addr \cup (reg - eip) \rightarrow s(x) = s'(x)$:
 Assume $s''(eip) = s(eip) + S(s) \wedge \exists y : y \in addr \cup (reg - eip) \wedge s''(y) \neq s(y)$:
 Assume $y \in addr$:
 $l = \emptyset$
 $\exists j : j \in l \wedge j = \langle \text{log_opcode_memwrite}, s(eip), x, s''(x) \rangle$
 \perp
 Assume $y \in (reg - eip)$:
 $l = \emptyset$
 $\exists j : j \in l \wedge j = \langle \text{log_opcode_regwrite}, s(eip), x, s''(x) \rangle$
 \perp
 \perp
 $s' = s''$
 Assume $s''(eip) \neq s(eip) + S(s) \wedge \forall y : y \in addr \cup (reg - eip) \rightarrow s(y) = s''(y)$:
 $l = \emptyset$
 $l = \langle \text{log_opcode_jump}, s(eip), s''(eip) \rangle$
 \perp
 $s = s''$
 Assume $s''(eip) = s(eip) + S(s) \wedge \forall y : y \in addr \cup (reg - eip) \rightarrow s(y) = s''(y)$:
 Assume $\exists z : s'(z) \neq s''(z)$
 Assume $z = eip$:
 $s'(eip) = s(eip) + S(s)$
 $s''(eip) = s(eip) + S(s)$
 $s'(eip) = s''(eip)$

$$\begin{array}{l}
\perp \\
\text{Assume } z \in \text{addr} \cup (\text{reg} - \text{eip}) \\
s'(z) = s(z) \\
s''(z) = s(z) \\
s''(z) = s'(z) \\
\perp \\
\perp \\
\neg \exists z : s'(z) \neq s''(z) \\
s' = s'' \\
s' = s'' \\
s' = s'' \\
\langle s, l \rangle \rightsquigarrow_P \langle s', l' \rangle \wedge \langle s, l \rangle \rightsquigarrow_L \langle s'', l' \rangle \rightarrow s' = s'' \\
\blacksquare
\end{array}$$

We can see from the above that the program traces generated by Logrind are indeed sufficient to recreate the state of an x86 executable at any given point in its execution *post-mortem* because all external inputs to the program are already known hence its behaviour may be considered fully deterministic. The above semantics may be trivially generalised to other processors and operating systems that Logrind may be ported to.

Appendix B

Testing schedule

The automatic tests are presented first. The specification field gives the specification the test is designed to check. The pass criterion specifies what the implementation needs to do to pass the test. The GDB commands field lists the commands that were used to check the pass criterion. The test program field specifies which test program was used with the GDB commands (one of A, B, C, or D below). The expected output field gives the expected output of said commands. The result field contains PASS if the expected outcome was obtained or FAIL and a description of the failure if not.

Test program A

```
1 void _start (void) {
2     int a = 1;
3     if (a) {
4         a = 2;
5     }
6     exit (0);
7 }
8
```

Test program B

```
1 #define __KERNEL_SYSCALLS__
2 #include <errno.h>
3 #undef __KERNEL_STRICT_NAMES
4 #include <linux/types.h>
5 #include <asm/unistd.h>
6 void myfunc (int b) {
7     b = 2;
8 }
9 void _start (void) {
10     int a;
11     a = 1;
12     myfunc (a);
13     sync ();
14     exit (0);
```

```
15 }
16
```

Test program C

```
1 void _start (void) {
2     int a;
3     int b;
4     a = 1;
5     b = a;
6     b = 3;
7     a = b;
8     b = a;
9     exit (0);
10 }
11
```

Test program D

```
1 void _start(void) {
2     int a = 0;
3     for (;;)
4         a++;
5 }
6
```

Test 1: "target logrind" command is available in GDB

Specification being tested:

The debugger must provide an option for launching the inferior under Logrind.

Pass criterion:

target logrind is a valid command.

GDB Commands:

help target logrind

Expected output:

Logrind program trace.

Result:

PASS

Test 2: Programs can be launched under Valgrind from GDB Specification being tested:

The debugger must make any commands that work on program traces available for use if a process is launched under Logrind in this manner.

Pass criterion:

target logrind history prints a program trace history.

GDB Commands:

```
target logrind
run
logrind history
```

Expected output:

```
1 (0.0083964): [1] \_init (???:0) rewrite esp: bffff4f8 => bffff4f4
```

⋮
Result:
PASS

Test 3: Program trace can be viewed after the inferior receives a signal. Specification being tested:

The debugger must make any commands that work on program traces available for use if a process is launched under Logrind in this manner.

Pass criterion:

target logrind history prints a program trace history.

GDB Commands:

```
target logrind
run
CTRL-C
logrind history 75
```

Expected output:

```
Program received signal SIGINT, Interrupt.
[Switching to process 19014]
_start () at progD.c:4
4      a++;
      push  %ebp
⋮
```

Result:
PASS

Test 4: Maximum database size (max-database-size) option is honoured. Specification being tested

The debugger should allow the user to set the maximum size of the program trace.

Pass criterion:

When the max-database-size option is set the database never exceeds this size.

Test program:

progD

GDB Commands:

```
set logrind max-database-size 16384
target logrind /tmp/autotest.db
info target
run
```

Expected outcome:

The size of the database file after 10 seconds is less than 16384 bytes.

Result:

FAIL

The Logrind skin did not respect the file size limit. There was not enough time to diagnose the source of this problem.

Test 5: "logrind history" command accepts the argument format: FILTER Specification being tested

The history command should apply a user-specified filter to its output.

Pass criterion:

Only rows matching the filter are output.

Test program:

progA

GDB Commands:

```
target logrind
run
logrind history addr2line(pc)='progA.c:6'
```

Expected outcome:

```
Program exited normally.
progA.c:6 exit (0);
:
```

Result:

PASS

**Test 6: "logrind history" command accepts the argument format: COUNT
Specification being tested**

The history command show allow a user to specify only a portion of the program trace to display.

The history command should only output events that match the filter.

Pass criterion:

Only COUNT rows are output.

Test program:

progA

GDB Commands:

```
target logrind
run
logrind history 10
```

Expected outcome:

10 and only 10 trace rows are output.

Result:

PASS

**Test 7: "logrind history" command accepts the argument format: -START
Specification being tested**

The history command show allow a user to specify only a portion of the program trace to display.

Pass criterion:

The first row output is START rows from the end of trace.

Test program:

progA

GDB Commands:

```
target logrind
run
logrind history -10
```

Expected outcome:

```
Program exited normally.
progA.c:6 exit (0);
    call    0x8048194
           84 (0.0249029): [1] _start + 31 (progA.c:6) regwrite esp: bffff5
:
```

(the program trace is 94 entries long)

Result:

FAIL The whole trace was printed rather than the last 10 lines. The problem was identified as a sour code line out of order and fixed.

Test 8: "logrind history" command accepts the argument format: START COUNT

Specification being tested

The history command show allow a user to specify only a portion of the program trace to display.

Pass criterion:

The first row output has sequence id START. Only COUNT rows are output.

Test program:

progA

GDB Commands:

```
target logrind
run
logrind history 10 5
```

Expected outcome:

```
Program exited normally.
10 (0.0100565): [1] call_gmon_start + 1 (???:0)
    regwrite ebp: bffff4f4 => bffff4e4
11 (0.0101694): [1] call_gmon_start + 3 (???:0)
    regwrite esp: bffff4e4 => bffff4e0
12 (0.0102894): [1] call_gmon_start + 3 (???:0)
    memwrite bffff4e0: 400142a8 = _rtld_global + 1188
13 (0.0104176): [1] call_gmon_start + 4 (???:0)
    regwrite ebx: _rtld_global + 1188 => call_gmon_start + 9
14 (0.0105445): [1] call_gmon_start + 10 (???:0)
    regwrite ebx: call_gmon_start + 9 => __JCR_LIST__ + 4
```

Result:

PASS

Test 9: "logrind history" command accepts the argument format: START COUNT FILTER

Specification being tested

The history command should apply a user-specified filter to its output.

The history command should only output events that match the filter.

The history command show allow a user to specify only a portion of the program trace to display.

Pass criterion:

The first row output has sequence id START. Only COUNT rows are output. Only rows matching the filter are output.

Test program:

progA

GDB Commands:

```
target logrind
run
logrind history 10 5 opcode=opcode_regwrite
```

Expected outcome:

```

Program exited normally.
10 (0.0092276): [1] call_gmon_start + 1 (???:0)
    regwrite ebp: bffff4f4 => bffff4e4
11 (0.0093629): [1] call_gmon_start + 3 (???:0)
    regwrite esp: bffff4e4 => bffff4e0
13 (0.0096593): [1] call_gmon_start + 4 (???:0)
    regwrite ebx: _rtld_global + 1188 => call_gmon_start + 9
14 (0.0098187): [1] call_gmon_start + 10 (???:0)
    regwrite ebx: call_gmon_start + 9 => __JCR_LIST__ + 4
15 (0.0099532): [1] call_gmon_start + 16 (???:0)
    regwrite esp: bffff4e0 => bffff4dc

```

Result:
PASS

Test 10: Program trace correctly reflects the operations performed by the program.

Specification being tested

The history command should show writes made by the program, jumps, function calls, function returns and system calls and the source line where these occurred. If read events are present in the program trace, the history command should show these as well.

Pass criterion:

The output should show one and only one write to a at line 11; one and only one call to myfunc at line 12; one and only one system call to sync at line 13; and one and only one read from a at line 12.

Test program:

```
progB
```

GDB Commands:

```

set logrind suppressions traceread.supp
target logrind
info target
run
logrind history

```

Expected outcome:

```

Program exited normally.
:
progB.c:11 a = 1;
    movl    $0x1,0xffffffff(%ebp)
:
      83 (0.0210219): [1] _start + 6 (progB.c:11)
        memwrite a: _end + 939226893 => 00000001
:
progB.c:12 myfunc (a);
:
    pushl  0xffffffff(%ebp)
:
      87 (0.0218213): [1] _start + 16 (progB.c:12)
        memread a: 00000001

```

```

    call    0x80481ec <myfunc>
    :
    94 (0.0226392): [1] _start + 19 (progB.c:12)
        call myfunc
    :
progB.c:13 sync ();
    call    0x8048222 <sync>
    :
    126 (0.0257782): [1] sync + 13 (unistd.h:366)
        syscall 36 = 808464432 (30303030)

```

Result:

FAIL Function calls, jumps and returns were not being traced by the Logrind skin. The problem was identified as a missing break statement and fixed.

Test 11: Source language expressions can be used as a filter.

Specification being tested

A filter may contain boolean expressions in the language of the program being debugged (e.g. `x > 5`)

Pass criterion:

Only rows which the boolean source language expression evaluates to true are output.

Test program:

```
progC
```

GDB Commands:

```
target logrind
run
logrind history 78 1 eval(sequence, 'a==3')
```

Expected outcome:

```

Program exited normally.
progC.c:8 b = a;
    mov    0xffffffffc(%ebp),%eax
    84 (0.0209832): [1] _start + 32 (progC.c:8)
        regwrite eax: 00000003 => 00000003

```

Result:

PASS

Test 12: Schema elements can be used in a filter.

Specification being tested

A filter may contain comparisons between one or more elements from the program trace schema (e.g. `opcode = opcode_jump`)

Pass criterion:

Only rows for which the comparison is true are output.

Test program:

```
progA
```

GDB Commands:

```
target logrind
run
logrind history opcode=opcode_memwrite
```

Expected outcome:

```

Program exited normally.
  2 (0.0085898): [1] _init (???:0)
      memwrite bffff4f4: bffff524 => bffff524
  6 (0.0090982): [1] _init + 6 (???:0)
      memwrite bffff4e8: _rtld_global + 1188 => _init + 11
  9 (0.0095159): [1] call_gmon_start (???:0)
      memwrite bffff4e4: bffff4f4 => bffff4f4
  ⋮ (only rows with opcode memwrite are output)

```

Result:

PASS

Test 13: Functions of schema elements can be used in filters.**Specification being tested**

A filter may contain functions of elements from the program trace schema as comparison operands (e.g. `addr2line(pc)="main.c:5"`)

Pass criterion:

Only rows for which the filter is true are output.

Test program:

progA

GDB Commands:

```

target logrind
run
logrind history addr2line(pc)="progA.c:4"

```

Expected outcome:

```

Program exited normally.
progA.c:4 a = 2;
  movl    $0x2,0xffffffffc(%ebp)
  79 (0.0246740): [1] _start + 19 (progA.c:4)
      jump _start + 19
  80 (0.0252202): [1] _start + 19 (progA.c:4)
      memwrite a: 00000001 => 00000002

```

Result:

PASS

Test 14: Program trace cursor can be set.**Specification being tested**

The cursor command should set the program trace cursor.

Pass criterion:

The show logrind cursor command returns the new cursor following a set logrind cursor command

Test program:

progC

GDB Commands:

```

target logrind
run
set logrind cursor 79
show logrind cursor

```

Expected outcome:

Program exited normally.

```

mov    0xffffffffc(%ebp),%eax
79 (0.0226014): [1] _start + 13 (progC.c:5)
regwrite eax: 00000000 => 00000001

```

The position of the trace cursor (0 = end of trace) is 79.

Result:

PASS

Test 15: Memory and register reads are relative to the program trace cursor.

Specification being tested

The debugger should return values for memory, registers and stack that they had at the cursor position.

Pass criterion:

The output of GDB's print command is correct for the current cursor position.

Test program:

progC

GDB Commands:

```

target logrind
run
set logrind cursor 79
print a
set logrind cursor 84
print a

```

Expected outcome:

```

Program exited normally.
progC.c:5 b = a;
mov    0xffffffffc(%ebp),%eax
79 (0.0227595): [1] _start + 13 (progC.c:5)
regwrite eax: 00000000 => 00000001

$1 = 1
progC.c:8 b = a;
mov    0xffffffffc(%ebp),%eax
84 (0.0238860): [1] _start + 32 (progC.c:8)
regwrite eax: 00000003 => 00000003

$2 = 3

```

Result:

PASS

Test 16: Program trace cursor can be switched on/off.

Specification being tested

The cursor command should allow the user to turn the program trace cursor off (memory, registers and stack are read from the inferior as usual).

Pass criterion:

When the program trace cursor is off memory values are read from the inferior. When the program trace cursor is on memory values are relative to the cursor.

Test program:

progC

GDB Commands:

```

target logrind
break 6

```

```

run
print b
break 9
cont
print b
set logrind cursor 81
print b
set logrind cursor 0
print b
set logrind cursor 81
print b
cont

```

Expected outcome:

```
[Switching to process 19184]
```

```
Breakpoint 1, _start () at progC.c:6
```

```
6  b = 3;
```

```
$1 = 1
```

```
Breakpoint 2 at 0x80481ca: file progC.c, line 9.
```

```
Breakpoint 2, _start () at progC.c:9
```

```
9  exit (0);
```

```
$2 = 3
```

```
progC.c:6 b = 3;
```

```
    movl  $0x3,0xffffffff8(%ebp)
```

```
    81 (0.0198289): [1] _start + 19 (progC.c:6)
```

```
        memwrite b: 00000001 => 00000003
```

```
$3 = 1
```

```
Program trace cursor is now off.
```

```
$4 = 3
```

```
progC.c:6 b = 3;
```

```
    movl  $0x3,0xffffffff8(%ebp)
```

```
    81 (0.0198289): [1] _start + 19 (progC.c:6)
```

```
        memwrite b: 00000001 => 00000003
```

```
$5 = 1
```

```
Program exited normally.
```

Result:

FAIL The output contained \$3 = 3 as opposed to \$3 = 1.

If the cursor was pointing at a memory write the post-image of that write was used instead of the pre-image. The cause was diagnosed as an incorrect SQL query which was subsequently fixed.

Appendix C

User guide

C.1 What is Logrind 2?

Logrind 2 is a tool for capturing program traces. A program trace is a list of all the things a process does including writing and reading to memory, jumps, system calls, etc.

C.2 How to install

You can download the most recent version of Logrind from:

<http://www.atomice.com/snapshots>

To compile from sources you should make and install sqlite before compiling valgrind or gdb.

C.3 How to start

Launch gdb as usual by typing `gdb filename` at a command line.

To activate logrind type `target logrind` at the GDB prompt. From now on GDB will trace any programs you run using Logrind. To run the program type `run` at the GDB prompt as usual.

When the process has exited or stopped due to a breakpoint you may examine the program trace using the `logrind history` command.

C.4 Logrind commands

C.4.1 `logrind history`

Syntax:

```
logrind history
```

```
logrind history COUNT
```

```
logrind history -START
```

```
logrind history START COUNT
```

```
logrind history START COUNT FILTER
logrind history FILTER
```

The `history` command displays a portion of the program trace. The region of the program trace that is displayed is controlled using the `START` and `COUNT` arguments. The `START` argument specifies the first line number to display. The `COUNT` argument specifies how many lines to display.

If the `START` argument is negative the last `START` lines of the program trace will be displayed.

The `FILTER` argument may be used to control which lines are displayed. For example to only show function calls you would use the filter `history /<call *>/`.

Example:

```
logrind history 1 10
```

```
1: [1] _init (???:0) regread ebp: bffff544
2: [1] _init (???:0) regread esp: bffff518
3: [1] _init (???:0) regwrite esp: bffff518 => bffff514
4: [1] _init (???:0) memwrite bffff514: bffff544 => bffff544
5: [1] _init + 1 (???:0) regwrite ebp: bffff544 => bffff514
6: [1] _init + 3 (???:0) regread esp: bffff514
7: [1] _init + 3 (???:0) regwrite esp: bffff514 => bffff50c
8: [1] _init + 6 (???:0) regwrite esp: bffff50c => bffff508
9: [1] _init + 6 (???:0) memwrite bffff508: 40013844 => 40232743
10: [1] _init + 6 (???:0) call call_gmon_start
```

Interpreting the program trace

A program trace line is interpreted as follows:

```
497: [1] main + 19 (hello.c:5)
      memwrite bffff528: 00000020 => 0000000e}

497      line number/cursor position
[1]      thread 1
main + 19  function name + offset
hello.c:5  source filename:line number
memwrite   opcode (one of memwrite, regwrite, memread, regread,
            jump, call, return, syscall, sysret)
bffff528  first argument to memwrite - the address written to
00000020  the pre-image (i.e. what was in memory at that address be-
            fore it was overwritten)
0000000e  the post-image (i.e. what was written to memory)
```

C.4.2 `logrind query`

Syntax:

```
logrind query QUERY
```

The `logrind query` command can be used to execute arbitrary SQL queries on the program trace database. The results will be returned in a table.

C.4.3 logrind sources

Syntax:

```
logrind sources
```

`logrind sources` command displays a list of valid program trace sources. The active source may be set with the `set logrind source` command.

C.4.4 logrind macro define

Syntax:

```
logrind NAME REPLACEMENT
```

```
logrind NAME ARG1, ..., ARGn REPLACEMENT
```

Defines a macro for use in a query or filter.

C.4.5 logrind macro undefine

Syntax:

```
logrind NAME
```

Undefines a previously defined macro.

C.4.6 set logrind cursor

Syntax:

```
set logrind cursor POSITION
```

The `set logrind cursor` command is used to move the program trace cursor. All GDB memory, register and stack commands operate relative to this cursor. GDB will behave as if your program had stopped at that point in the process' execution because of a breakpoint.

If `POSITION` is positive the cursor will be set to that absolute line number of the program trace. If `POSITION` is negative the cursor will be set to `POSITION` lines from the end of the trace. If `POSITION` is 0 the program cursor will be switched off.

Example:

```
set logrind cursor 100
```

C.4.7 show logrind cursor

Syntax:

```
show logrind cursor
```

Shows the current position of the program trace cursor.

C.4.8 logrind cursor backwards

Syntax:

```
logrind cursor backwards
```

```
logrind cursor backwards COUNT
```

The `logrind cursor backwards` command moves the program trace cursor backwards `COUNT` rows. If the `COUNT` argument is omitted the cursor is moved back one row.

C.4.9 `logrind cursor forward`

Syntax:
`logrind cursor forward`
`logrind cursor forward COUNT`

The `logrind cursor forward` command moves the program trace cursor forward `COUNT` rows. If the `COUNT` argument is omitted the cursor is moved forward one row.

C.4.10 `set logrind source`

Syntax:
`set logrind source ID`

Each time you run your program using Logrind a new program trace is generated. The `set logrind source` command is used to change which program trace is currently active.

C.4.11 `show logrind source`

Syntax:
`show logrind source`

Show the id of the currently active source.

C.4.12 `set logrind filter`

Syntax:
`set logrind filter`
`set logrind filter FILTER`

The `set logrind filter` command sets a default filter to use with the `logrind history` command if no `FILTER` argument is supplied.

If the command is used without any arguments no default filter is used.

A filter is an SQL boolean expression.

C.4.13 `show logrind filter`

Syntax:
`show logrind filter`

Shows the default filter for the `logrind history` command.

C.4.14 set logrind suppressions

Syntax:

```
set logrind suppressions FILENAME
```

Tells the Logrind 2 skin to use the given suppressions file when launching applications. This command only works before the `target logrind` command.

C.4.15 show logrind suppressions

Syntax

```
show logrind suppressions
```

Shows the filename of the suppressions file currently in use.

C.4.16 set logrind max-database-size

Syntax:

```
set logrind max-database-size SIZE
```

Sets the maximum size of the program trace database to `SIZE` bytes. This command only works before the `target logrind` command.

C.4.17 show logrind max-database-size

Syntax:

```
show logrind max-database-size
```

Show the maximum size of the program trace database.

C.4.18 target logrind

Syntax:

```
target logrind
```

```
target logrind DATABASE
```

Use the `target logrind` to switch on program trace capture.

Logrind will automatically create a new program trace database if you do not supply one.

C.5 Query syntax

Queries in Logrind are extended SQL queries. A number of program trace specific functions are available for use both in queries and filters. A query is a complete SQL command, whereas a filter is an SQL boolean expression. Anything that can appear after `where` in an SQL `select` statement is a valid filter.

C.6 Functions

eval(sequence, 'expr')

Evaluates a source language expression at a particular point in the program trace.

addr2line(pc)

Returns the source filename and line numbers corresponding to the given program counter address.

line2addr('file:line')

Returns the start address of the given line.

symbol2beginaddr(sequence, 'symbol')

Returns the start address of the given symbol.

symbol2endaddr(sequence, 'symbol')

Returns the end address of the given symbol.

addr2symbol(sequence, address)

Returns the name of the symbol at the given address plus some offset.

inscope(sequence, 'symbol')

Returns true if the given symbol is in scope else false.

current_cursor()

Returns the current cursor position.

current_source()

Returns the id of the active source.

base64decode2int('data')

Converts the given base 64 data to an integer.

base64decode2uint('data')

Converts the given base 64 data to an unsigned integer.

hex2int('hex')

Converts the given hexadecimal number to an integer.

hex2uint('hex')

Converts the given hexadecimal number to an unsigned integer.

int2hex('hex')

Converts the given integer to a hexadecimal number.

uint2hex('hex')

Converts the given unsigned integer to a hexadecimal number.

C.7Suppressions

You may set a suppressions file using the `set logrind suppressions` command. A suppressions file specifies what program trace events you want to be recorded. The contains one or more blocks in the following format:

```
{
  All libraries
  Logrind: All
  obj:/lib/*
}
```

The first line is a label or comment for the suppression and is ignored.

The second line specifies what events are to be suppressed. The permitted values are:

all, register read, register write, memory read, memory write,
jump, call, return, system call, general register write, stack
frame register write, general register read, stack frame register
read, read, register, memory

Multiple values may be separated using commas. Prefixing any value with the allow keyword explicitly allows traces of that type to be recorded, overriding any previous suppressions.

See the `logrind.supp` file for more examples.