Tralfamadore: Unifying Source Code and Execution Experience

(Short Paper)

Geoffrey Lefebvre, Brendan Cully, Michael J. Feeley, Norman C. Hutchinson and Andrew Warfield

Department of Computer Science, University of British Columbia, Vancouver, Canada

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids, Tracing; D.2.6 [*Programming Environments*]: Interactive Environments; H.3.3 [*Information Search and Retrieval*]: Information filtering

General Terms Design, Experimentation, Languages

Abstract

Program source is an intermediate representation of software; it lies between a developer's intention and the hardware's execution. Despite advances in languages and development tools, source itself and the applications we use to view it remain an essentially static representation of software, from which developers can spend considerable energy postulating actual behavior.

Emerging techniques in execution logging promise to provide large shared repositories containing high-fidelity recordings of deployed, production software. Tralfamadore[1] is a system that combines source and execution trace analysis to capitalize on these recordings, and to expose information from the "experience" of real execution within the software development environment, allowing developers to inform their understanding of source based on how it behaves during real execution.

1. Introduction

"What were they thinking?"

This question, posed in one of several possible intonations, is often a developer's first reaction to unfamiliar source code. As they gain familiarity with a large and complex code base—such as an operating system kernel—they become better able to infer the expected behavior of source code and to safely change it. Still, the relationship between a developer and source code is frequently interrogatory, often requiring instrumentation or debuggers to answer questions such as: "What modifies this data structure?" or "What locks are held when this function is called?"

The aim of the work described in this paper is to enhance conventional tools that are used to interact with source code in order to provide developers with a sense of the *experience* of executing that source. By embedding execution details directly within a source browser, we allow developers to see the bigger picture; they are able to better understand things like the frequency with which specific regions of code run, the ranges of data that are processed, and the flow of control through source.

Our intention is not simply to provide static annotations, such as the call frequencies reported by a statistical profiling tool. Instead, we approach program understanding and debugging as an online query and analysis problem where a view of program source may be used to specify constraints, such as specific control flow paths or data values, that refine the presentation of that source. Unlike a statistical profiler, the annotations are a result of online queries and analyses of complete execution traces applied to source views. Unlike conventional debuggers, these analyses apply to existing execution traces, and are able to summarize very large numbers of executions, rather than focusing on a single (and generally contrived) execution context.

The system we describe gathers detailed execution traces associated with a specific source version and stores them in a central location where they are analyzed and indexed. Developer tools then interact with these traces by querying for relevant portions of execution and then performing dynamic analysis on them in order to adjust the presentation of program source to the developer. For example, a source browser might annotate a function like the one shown in Figure 3 to summarize the specific control paths taken through it during traced execution. This trace immediately assists the developer by allowing them to differentiate common from exceptional paths of execution. Furthermore, the developer may focus their view of the source by selecting a specific control flow path, collapsing the view to only show the lines executed in that path.

[1] In [Kurt Vonnegut's] Slaughterhouse-Five, Tralfamadore is the home to beings who exist in all times simultaneously, and are thus privy to knowledge of future events, including the destruction of the universe at the hands of a Tralfamadorian test pilot. – *Wikipedia*

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.
Copyright © 2009 ACM 978-1-60558-482-9/09/04. . . $5.00

How is this data structure used?	***What is this function for?***
What reads and writes it?	What calls it? (both immediate and higher-level callers)
How frequently is it accessed?	How frequently is it called?
How much memory does it consume over time?	What control flow paths are taken through it?
What values does it take?	What values does it return?
What are the semantics of access (e.g., ordered array access, atomic updates to all fields in a struct)?	Are calls to it correlated with specific data structures, like locks?

Table 1. Some common questions asked in attempting to understand program source.

2. Understanding Source Code

Table 1 lists a set of questions that may be useful in attempting to understand how source code behaves. Currently, there are three broad classes of tools for answering this kind of query: fine-grained tools, like debuggers, that assist in deconstructing a single execution context; coarse-grained tools, like statistical profilers, that summarize the high-level behavior of an application; and static analysis tools, which work directly with source, independent of actual execution.

Fine-grained and interactive analysis tools, such as debuggers, allow a developer to directly interact with running software. These tools have the benefit of exposing complete system state and allowing memory to be read and written. However, they represent only a single context of a program's execution as it progresses through a single run. It is often very difficult to attach a debugger to exceptional points in program execution, especially where these points involve environmental factors such as long run times or external dependencies like network connections. Although time-travelling debuggers [King 2005, O'Callahan 2008] allow a programmer to move along the time axis in either direction, their view is still constrained to one instant at a time. Program slicing techniques [Weiser 1981] help automate the discovery of causal relationships between statements for a particular execution context but provide little intuition about the behaviour of the code across a broad range of inputs.

Coarse-grained summarization tools, provide aggregate information about application behavior. For instance, the GNU profiler uses program instrumentation to record call frequencies at function granularity. Other tools make use of hardware performance registers to provide instruction-level execution frequencies in order to allow developers to better understand where execution time is actually being spent. Dynamic binary analysis tools such as Valgrind [Nethercote 2007] perform run-time analysis of application execution to find problems like memory leaks or inefficient cache usage. While these tools are clearly helpful, they require an *a priori* understanding of what analysis should be performed, and produce summarized reports that are unable to answer follow-up questions.

Static analysis of application source is used by a number of developer aids, from navigational and source browsers to more complicated checking tools that validate the correct use of locks [Engler 2003]. While static tools can perform sophisticated source analysis, they rarely incorporate information from actual execution. As a result, they have trouble assessing the relevance of the wealth of information that they are capable of producing.

These three classes of tools are each helpful for understanding source, but they are all inefficient in some way. Debuggers require that developers spend considerable amounts of time in uninteresting execution states while trying to find the interesting ones. More broadly, all of these tools leave developers cyclically performing a series of debugging or analysis experiments over numerous independent runs even though they are not changing the source.

2.1 Tralfamadore

Tralfamadore uses detailed execution trace data in an attempt to unify the above three classes of tools. Our intention is to present a view of source code—what a program *should* do—superimposed with trace data and analysis, or what it actually does.

Our system has two important high-level goals:

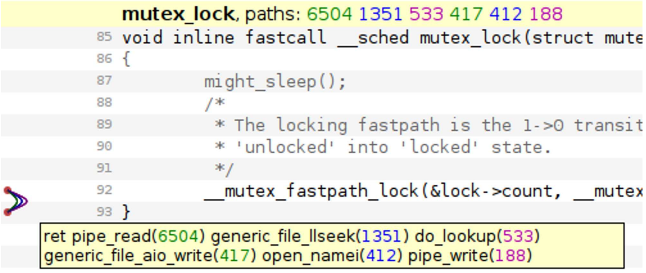
1. ***Simultaneously present the recorded execution of software from many points in time.*** Tralfamadore aims to preserve the detailed system view afforded by a debugger, while also representing the collective execution of the system over a long period of time. In other words, it allows developers to become "unstuck in time", presenting fine-grained analysis throughout many points in the execution history.
2. ***Allow the developer to interactively refine their view of execution.*** High-fidelity trace data should allow the developer to overcome the inefficiencies of cyclical debugging, letting them phrase follow-up question as refinements of the scope of execution being examined which included progressively more detail. In short the developer should be able to "drill down" in order to better understand specific nuances of system behavior. In the extreme, the developer should be able to refine their view of trace data to a single execution context at a single point in time, and ask the system to regenerate an instance of that system, potentially attached to a conventional debugger.

3. Examples

This section demonstrates Tralfamadore's ability to present the detailed and complex information resulting from trace analysis in an intuitive manner. Our prototype is in its infancy but can already reveal several interesting properties, as we demonstrate using the Linux source tree.

3.1 Function and Data Users

A major challenge to understanding the way that an individual function or data structure is used is in identifying the code that uses it. Static analysis, or even simpler techniques,[2] are useful, but hardly sufficient. First, these tools are unable to follow indirection. Second, they do not provide any insight into the relative frequency of access, making it difficult for developers to start with the "common case".



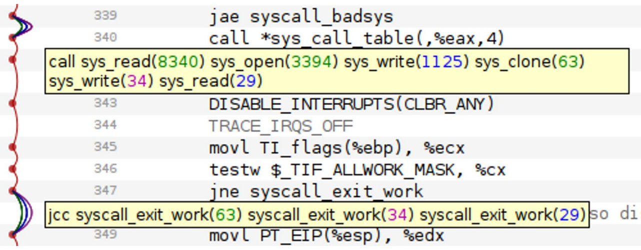
```
                mutex_lock, paths: 6504 1351 533 417 412 188
85 void inline fastcall __sched mutex_lock(struct mute
86 {
87         might_sleep();
88         /*
89          * The locking fastpath is the 1->0 transit
90          * 'unlocked' into 'locked' state.
91          */
92         __mutex_fastpath_lock(&lock->count, __mutex
93 }
   ret pipe_read(6504) generic_file_llseek(1351) do_lookup(533)
   generic_file_aio_write(417) open_namei(412) pipe_write(188)
```

Figure 1. Some users of the mutex_lock function.

Figure 1 shows the Linux mutex_lock function annotated with trace data. Locking semantics aside, this function represents the common idiom of an accessor function guarding a specific variable, in this case a mutex. The annotated code immediately provides the developer with three useful pieces of information: First, while mutex_lock is called over 8000 times in the trace, the slow path is never taken; if it were, a box highlighting a call on line 52 to __mutex_lock_slowpath[3] would be shown. Second, it is called by six different callers, resulting in independently colored control flow tags in the annotation at the top of the function. Finally, the actual frequencies of each of these calling contexts is reported in the box at the bottom of the function which shows where control returns to; this allows the developer to immediately focus on pipe_read as the most frequent caller.

3.2 Control Flow Indirection

In the example above, we revealed the users of a function. In many cases, the inverse operation can also be very useful. When calls are performed indirectly, e.g., through function pointers, static tools are easily stymied. While dynamic analysis does not necessarily present a complete inventory of control flow targets for a given site, it does allow detailed insight into real invocations.



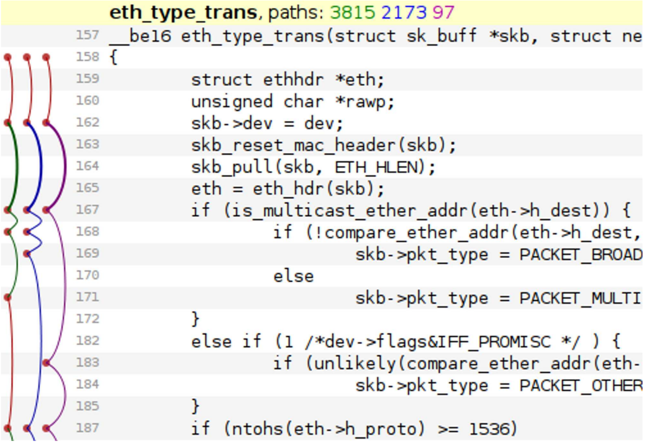
```
339         jae syscall_badsys
340         call *sys_call_table(,%eax,4)
   call sys_read(8340) sys_open(3394) sys_write(1125) sys_clone(63)
   sys_write(34) sys_read(29)
343         DISABLE_INTERRUPTS(CLBR_ANY)
344         TRACE_IRQS_OFF
345         movl TI_flags(%ebp), %ecx
346         testw $_TIF_ALLWORK_MASK, %cx
347         jne syscall_exit_work
   jcc syscall_exit_work(63) syscall_exit_work(34) syscall_exit_work(29) so di
349         movl PT_EIP(%esp), %edx
```

Figure 2. Following an indirect call in sysenter_entry.

Figure 2 shows one of Linux's system call entry points, sysenter_entry. On line 340, the system call number in register EAX is used as an offset into a jump table, in an assembler invocation that is difficult to analyze statically. Tralfamadore annotates this statement with a list of the jump targets that are taken in the trace. As with the accessor example above, it clearly presents a ranked list of system calls, providing the developer with an intuitive sense of the common uses of the underlying code.

Because it uses actual traces, it can display useful information that is not available from profile-based tools. For example, it is easy to see that of the 4 system calls invoked in this slice of trace data, one (sys_clone) always calls syscall_exit_work after it returns, one (sys_open) never calls it, and two (sys_read, sys_write) sometimes do but usually don't. In Section 4.3, we discuss how this execution context may be used to produce a refined view of program flow satisfying non-local conditions.

3.3 Path Analysis



```
           eth_type_trans, paths: 3815 2173 97
157 __be16 eth_type_trans(struct sk_buff *skb, struct ne
158 {
159         struct ethhdr *eth;
160         unsigned char *rawp;
162         skb->dev = dev;
163         skb_reset_mac_header(skb);
164         skb_pull(skb, ETH_HLEN);
165         eth = eth_hdr(skb);
167         if (is_multicast_ether_addr(eth->h_dest)) {
168                 if (!compare_ether_addr(eth->h_dest,
169                         skb->pkt_type = PACKET_BROAD
170                 else
171                         skb->pkt_type = PACKET_MULTI
172         }
182         else if (1 /*dev->flags&IFF_PROMISC */ ) {
183                 if (unlikely(compare_ether_addr(eth-
184                         skb->pkt_type = PACKET_OTHER
185         }
187         if (ntohs(eth->h_proto) >= 1536)
```

Figure 3. Processing 3 packet types in eth_type_trans.

In large functions, the set of possible paths through the code can quickly become obscured by a cascade of complex conditional jumps–this number is typically far less than $2^{conditionals}$ due to inter-branch dependencies. By presenting the set of actual paths taken, Tralfamadore makes it much easier to understand these dependencies. As a simple example, consider the eth_type_trans function in Figure 3.

[2] i.e., grep

[3] __mutex_lock_slowpath is the second parameter on line 52 and unfortunately runs off the edge of the figure.

We can clearly see that there are three distinct control flows corresponding to the type of ethernet packet being processed (multicast, broadcast, or normal). The distribution of these packet types during the trace is also presented, allowing a developer to better understand the actual workload being inspected. For instance, in this example 5,988 packets are multicast or broadcast versus only 97 normal packets, implying that the host is engaged in relatively little active network communication during the trace period.

4. System Architecture

Figure 4 presents an overview of Tralfamadore, and details how it is currently configured with regard to the analysis examples shown in the previous section. Tralfamadore is divided into three major components. The *Execution Trace Facility* is responsible for recording execution and generating a persistent log. The *Backend Analysis Engine* performs streaming transformations on the trace data, reconstructing system state and mapping it back onto program source. Finally, a *Client Interface* interacts with both the analysis engine and the program source repository to present information to the developer.

The current system has focused exclusively on the recording and analysis of the Linux kernel. As a complex piece of multi-threaded low-level software, Linux is an excellent target for Tralfamadore. That said, the system is hardly limited in its application to OS kernel code: we intend to extend it to include application code, and to present source annotations for languages other than C.

4.1 Execution Trace Facility

The work in this paper is unconcerned with the efficient capture and indexing of execution traces. A number of projects already exist that aim to capture high-fidelity execution traces in hardware [Xu 2003], software [Bhansali 2006] and at the virtualization layer [Dunlap 2002]. Recent work has demonstrated that the deterministic event logging and replay systems in a commercial hypervisor may be used to decouple trace capture from trace analysis and result in a trace collection overheads averaging 5% on common workloads [Chow 2008, Xu 2007]. Our prototype execution trace facility is a modified version of the QEMU emulator [Bellard 2005].

The system produces two parallel trace components. First, an *Instruction Translation Table* is extended whenever the emulator translates a new basic block of program binary. This table maps the current instruction pointer (EIP) to the in-memory instruction that is actually emulated, and serves two purposes. First, it reduces the size of the execution trace, by only requiring complete instructions to be stored once and referenced by EIP. Second, it allows the system to handle changes to the executing binary (e.g., self-modifying code) that occur during execution. Self-modifying code has become increasingly prevalent in modern systems; the Linux kernel, for instance uses it to tune lock implementations to specific system configurations, and to adapt a single OS kernel to specific virtual and physical boot environments.

The second component produced by QEMU is the *Execution Trace Log*. This log identifies all instructions that are executed and their associated side effects to memory and register state. It also contains details of events such as interrupts and exceptions. These two traces are then merged into a single *Detailed Trace*, which is the complete execution log of the system. Splitting the initial trace into two components is largely a matter of efficiency. Because QEMU stores translated instructions in a code cache for performance, the instruction trace grows at a rate much slower than the execution trace. We have observed more than two orders of magnitude difference between the two.

It is worth mentioning that the current QEMU-based implementation is the second prototype execution trace facility that we have implemented. Our early prototype took advantage of the branch trace store (BTS) feature that has been available on Intel processors since the Pentium 4 [Intel 2008], allowing the generation of a continuous log of all taken branches. Branch trace information alone was insufficient to perform many dynamic analyses, and handling self-modifying code in particular would have meant extending the implementation to track modifications to code pages. We measured the baseline overhead of BTS to be a 20-30x slowdown on the system, *before* adding the extensions required to achieve comprehensive traces. The QEMU implementation is currently comparable to that of BTS, but incurs considerable overhead in writing out trace data. We have found the emulator-based approach to be faster to extend, and generally more efficient than the processor feature.

4.2 Trace Analysis

The core of our system is a streaming trace analysis engine, which allows a pipeline of dynamic analysis modules to be applied to trace data. This approach allows us to quickly develop and test new analysis components, and to process very large traces without requiring large memory overheads. We plan to extend the system to parallelize trace analysis across a cluster of servers, eventually providing a scalable analysis engine for large software systems.

Our analysis engine is currently implemented in approximately 4500 lines of OCaml. It uses the trace we generate from QEMU but could be easily adapted to use traces from other tracing environments such as Nirvana [Bhansali 2006] or Retrace [Xu 2007]. The trace data is read from disk and converted to an internal representation that is passed through the individual pipeline stages. Stages process trace data, and are able to both annotate the stream with additional metadata and to build in-memory data structures, such as caches, as look-up services and optimizations for later stages.

For the examples shown in Section 3, in which the system is analyzing an execution trace of the Linux kernel during a kernel compilation, Tralfamadore's pipeline is configured with the four stages illustrated in Figure 4. Analyzing the

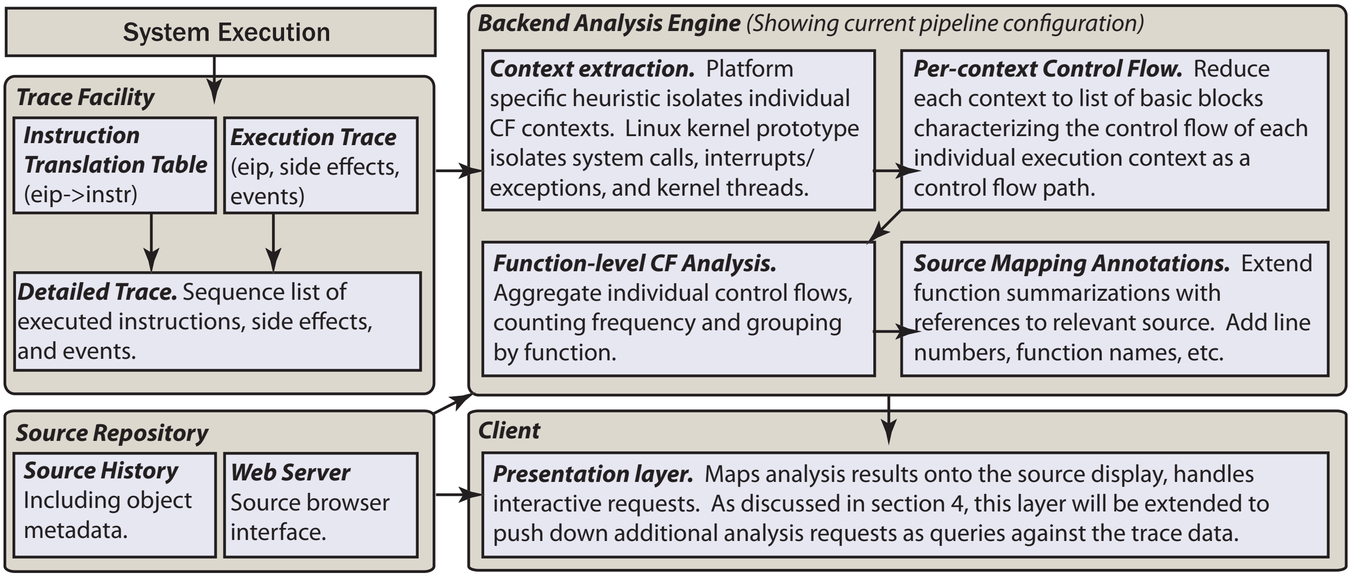

System Execution
Trace Facility
Instruction Translation Table (eip->instr)
Execution Trace (eip, side effects, events)
Detailed Trace. Sequence list of executed instructions, side effects, and events.
Source Repository
Source History Including object metadata.
Web Server Source browser interface.
Backend Analysis Engine (Showing current pipeline configuration)
Context extraction. Platform specific heuristic isolates individual CF contexts. Linux kernel prototype isolates system calls, interrupts/ exceptions, and kernel threads.
Per-context Control Flow. Reduce each context to list of basic blocks characterizing the control flow of each individual execution context as a control flow path.
Function-level CF Analysis. Aggregate individual control flows, counting frequency and grouping by function.
Source Mapping Annotations. Extend function summarizations with references to relevant source. Add line numbers, function names, etc.
Client
Presentation layer. Maps analysis results onto the source display, handles interactive requests. As discussed in section 4, this layer will be extended to push down additional analysis requests as queries against the trace data.

Figure 4. Trace analysis pipeline.

trace of a running OS is challenging, as it contains many concurrent execution contexts, such as system calls or interrupts which may occur at any given point in time. The analysis system must start with the system-granularity trace stream, and refine the representation up to the function-granularity annotations demonstrated in Section 3.

In the first stage, the system performs *context extraction* to isolate individual flows of execution. This stage encodes a heuristic that tracks task switch, interrupt, and exception events as implemented by Linux.[4] It then annotates the trace stream with framing information to label the individual extents of execution associated with each execution flow. The second stage performs *per-context control flow* by concatenating the extents of each individual labelled flow and constructing a list of the basic blocks that make up that context's flow. At the end of this stage, the execution trace has been reduced into a list of all individual control flows through the system, each described as a series of basic blocks.

The remaining two stages work upwards from this decomposition. First, a *function-level control flow analyzer* aggregates the individual control flows associated with each function, and builds per-function control flow trees. Each tree describes the set of unique control flows through a given function, and annotates each flow with a frequency count. The final stage takes the resulting trees and performs *source mapping annotations* using DWARF debugging information to further annotate the trees with information like source line numbers and symbol names.

4.3 Source Repository and Client

Tralfamadore presents annotated source listings to developers through a web-based source browser. We have modified the source browser interface provided by the Mercurial [Mackall 2005] version control system to associate trace data with the source that produced its executable. The trace viewer adds trace annotations to the source file viewer, including the lines comprising each basic block, the order of execution of basic blocks, and the targets of branches, calls, returns and so on. In our current prototype, the server provides precomputed summarizations of program flow (which result from the final stage of the analysis pipeline above) along with client-side javascript to visualize it.

To facilitate the interactive exploration of execution histories, most of the work of visualization is performed on the client side, within the user's web browser. The client renders graphs of execution based on the raw data about basic blocks and branches taken, as supplied by the server-side extension. The user can filter for particular execution paths by selecting one of the set of paths or selecting those where function pointers take on particular values.[5] For example, in Figure 3, the user could choose to display only the path taken by multicast packets by selecting the branch passing through line 171. Similarly, in Figure 2, the user could distinguish invocations of the same system call that end up scheduling further work before returning from those that don't.

5. Future Work and Conclusions

Tralfamadore's design hypothesizes that it will soon be reasonable to build central repositories containing detailed

[4] The current heuristic uses interrupts, exceptions, and traps to mark the start of a context, and iret/sysexit to mark the end. We use a Linux-specific heuristic of tracking the esp0 field of the TSS to detect context switches.

[5] In the current prototype we only track control flow data such as function pointers; eventually we will track all data changes and will support more extensive filtering capabilities.

recordings of program execution, and explores how these can be used to assist developers in understanding and improving software. Even with current techniques, we believe that this approach can be usefully applied, for instance, to record regression test runs of software releases as a centralized tool for developers.

While the majority of related work has been discussed in earlier sections of the paper, it is worth mentioning that recent work in programming languages research has explored building query languages that simplify the encoding of dynamic analyses [Martin 2005, Goldsmith 2005]. These systems have the same operational limitations of other binary analysis tools (applications must be rerun to apply a new query, for instance), but provide an elegant high-level interface to evaluate execution. We hope to extend these approaches in the future to apply to whole-system traces and to simplify the development of new analysis extensions.

The current prototype presents an end-to-end implementation of such a system, from execution recording, through analysis, to presentation as an annotated interactive source browser. Developers using the system are “unstuck in time” and able to immediately visualize huge amounts of execution as it pertains to individual areas of source. An immediate area of development involves extending our analysis engine to track the state of data in addition to control flow, allowing us to better answer questions from the first column of Table 1. We are extending the client to allow new queries to be issued to the backend, and to have partial results reported and displayed in an online manner as the trace is processed.

The current prototype is intended to act as a platform for a considerably more general execution analysis system. As Tralfamadore matures, we hope to be able to perform more complex analysis tasks, including the identification of outlying and exceptional execution states which may represent a source of either bugs or attacks. We also hope to allow developers to retroactively state assertions regarding the execution of their systems, and to validate these assertions against execution traces. With assertions, this would allow a popular defensive programming technique to be applied retroactively and without the need for cyclic re-compilation and re-execution as assertions are added or changed.

Tralfamadore is in its infancy, but we believe it demonstrates the power of its approach to code analysis. When real execution history is overlaid directly upon the source code that produced it, the gap between intention and effect becomes narrow enough to be bridged. To experiment with a (sometimes) live version of the system, point your browser at `http://tralfamadore.cs.ubc.ca/`.

References

[Bellard 2005] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[Bhansali 2006] S. Bhansali, W. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, 2006.

[Chow 2008] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, 2008.

[Dunlap 2002] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation*, 2002.

[Engler 2003] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[Goldsmith 2005] S. Goldsmith, R. O’Callahan, , and A. Aiken. Relational queries over program traces. In *Proceedings of the ACM SIGPLAN 2005 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[Intel 2008] Intel. Intel 64 and ia-32 architectures software developer’s manual volume 3b: System programming guide. http://www.intel.com/products/processor/manuals/, 2008.

[King 2005] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, 2005.

[Mackall 2005] M. Mackall. Mercurial. http://www.selenic.com/mercurial/, 2005.

[Martin 2005] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[Nethercote 2007] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[O’Callahan 2008] R. O’Callahan. Chronomancer: C/C++ trace-based debugger based on chronicle and eclipse. http://code.google.com/p/chronomancer/, 2008.

[Weiser 1981] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[Xu 2003] M. Xu, R. Bodik, and M. D. Hill. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[Xu 2007] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Third Annual Workshop on Modeling, Benchmarking and Simulation, held in conjunction with the 34th Annual International Symposium on Computer Architecture*, 2007.