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

Abstract
Program source is an intermediate representation of software; it lies between 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 behaviour.

Emerging techniques in execution logging promise to provide large shared repositories containing high-fidelity recordings of deployed, production software. Tralfamadore\footnote{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} 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, subject to a number of different intonations, is a frequent initial reaction when working with unfamiliar program source code. As a developer gains 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 interrogational, 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 worked with, and the expected 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 are in turn used to refine the presentation of that source. Unlike a statistical profiler, the annotations are a result of online queries and analyses of 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 these traces 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 it in order to adjust the presentation of program source to the developer. For example, a source browser might annotate a function, as shown in Figure 3, to summarize the specific control paths taken through that function during traced execution. This trace immediately assists the developer by allowing them to differentiate common from exceptional paths of execution. Further, the developer may constrain their view of the source by selecting a specific control flow path, and collapsing their view to only show the relevant, associated source lines.

The remainder of this short paper further motivates the need to provide developers with a view of execution that is broader and more efficient to use than what is available from current tools. It then describes a prototype implementation that demonstrates the effectiveness of exposing information from program execution directly within a source browser.
2. Understanding Source Code

Table 1 lists a set of questions that are usefully asked in attempting to understand how source code behaves. These questions, or others like them are the first thing that developers ask in attempting to work with existing source. Currently, there are three broad techniques that can be brought to bear in source understanding: 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 finally static analysis-based tools, which work directly with source, independent of actual execution.

### Fine-grained and interactive analysis tools

Such tools, like debuggers, allow a developer to directly interact with running software. These approaches 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 specific interactions with external sources such as clients on the network. Although time-travelling debuggers [10, 1] 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 [14, 3] help automate the discovery of causal relationships between statements for a particular execution context but provide little intuition about the behavior of the code across a broad range of inputs.

### Coarse-grained summarization tools

Such tools, like statistical profilers, provide aggregate information on how an application behaves. The GNU profiler, for instance, uses program instrumentation to record call frequencies at a function granularity. Intel’s VTune and OProfile capitalize on 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 [12] allow runtime evaluations of execution to produce reports on properties such as leaked memory or inefficient cache usage. While these tools are clearly helpful, they require an a priori understanding of what analysis is to be performed, and result in summarized reports that are of limited value in answering follow-up questions.

**Static analysis** of application source is used by a number of understanding tools, from navigational and source browsers such as Ctags and LXR, to more complicated checking tools that validate the correct use of locks [9] or security properties [13]. While static tools are incredibly thorough at analyzing source, they rarely incorporate information from actual execution. As such, they are of limited value in prioritizing the relevance of the wealth of information that they are capable of producing.

While these three loose classes of tools each provide individual benefits in understanding and working with source, they are all incredibly inefficient to use. 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 are iterative in nature: developers are left cyclically performing a series of experiments that involve debugging or analyzing an application 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 debug-

---

Table 1. Common questions asked in attempting to understand program source.
ging, as follow-up questions may be phrased as progressively more detailed views that refine the scope of execution being examined. 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 a running instance of that system, potentially for use under a conventional debugger.

3. Examples
This section demonstrates the ability of Tralfamadore to present the detailed and complex information that results from trace analysis in an intuitive manner. The current prototype is in the early stages, as will be discussed in Section 4, but already reveals several interesting aspects of the Linux source tree.

3.1 Function and Data Users
A major challenge in 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”.

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 to mutex_lock_slowpath 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.

\(^2\text{i.e., }\text{grep}\)

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, source-level tools alone 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.

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
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 \(2^{\text{conditionals}}\) 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. 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 that is accessible to the rest of the system. The Backend Analysis Engine performs streaming transformations of the trace data, reconstructing system state and mapping that state 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 largely 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 [15, 6], software [5] and at the virtualization layer [8, 7, 16]. The ability to generate execution traces with low-overhead is already available in at least one commercial virtualization product, and we believe that in the near future it will be completely feasible to record replayable execution traces of deployed enterprise software.

Unfortunately for the purposes of this research, the lack of a freely available trace facility meant that we had to build our own in order to collect execution logs. Our current execution trace facility is a modified version of the QEMU emulator [4].

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 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 the details of events such as interrupts and exceptions as they occur during execution. 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 [2], allowing the generation of a continuous log of all taken branches. Branch trace information alone was entirely 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 develop on, 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 has the immediate benefits of allowing 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 [5] or Retrace [16]. The trace data is read from disk and converted to an internal representation that is passed through the individual pipeline stages. The stages then 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 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 of analysis, the system performs context extraction to isolate individual flows of execution within the trace. This stage encodes a heuristic that tracks task switch, interrupt, and exception events as implemented by Linux. 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 the object and DWARF data to further annotate the trees with information like source line numbers and symbol names.

### 4.3 Source Repository and Client

Tralfamadore currently presents annotated source listings to developers through a web-based source browser. We have modified the source browser interface provided by the Mercurial [11] version control system to associate trace data with the specific revision of 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

---

1 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.
pointers take on particular values.\textsuperscript{4} 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 is based around the idea that it will soon be reasonable to build central repositories that contain detailed recordings of program execution, and explores how these recordings can be used to assist developers in understanding and improving software. Even if low-overhead tracing is not realizable in the immediate term, we believe that this technique can be usefully applied, for instance, to record regression test runs of software releases as a centralized tool for developers.

The current prototype realizes 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. One 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 the first column of questions shown in Table 1. Secondly, we are extending the client to allow new queries to be pushed down 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 considerably more complex analysis tasks, including the identification of outlying and exceptional execution states which are likely worth understanding as a source of either bugs or attacks. We also hope to take advantage of the ability to push new analyses down the backend as a means to allow developers to retroactively state assertions regarding the execution of their systems. Assertion statements are common in both debugging and defensive programming, but as with other techniques involve cyclic debugging. By allowing the application of assertions retrospectively, developers can use Tralfamadore to validate their assumptions about the behavior of a system.

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.nssl.cs.ubc.ca/.

References


\textsuperscript{4}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.