Debugging through Time with the Tralfamadore Debugger

Christopher C. D. Head Geoffrey Lefebvre Mark Spear Nathan Taylor Andrew Warfield
University of British Columbia
{chead,geoffrey,mspear,tnathan,andy}@cs.ubc.ca

Abstract

Program debugging has almost universally been considered from the perspective of performing detailed examination of a single program target (application, operating system, etc.) at a single point in time. We present an early prototype of the *Tralfamadore Debugger (TDB)*, a software debugger based on the *Tralfamadore* offline dynamic analysis engine. Unlike conventional debuggers, TDB presents a source-level debugging interface on a CPU-level execution log. The system maps processor-level events back to source-level semantics and allows developers to examine all of execution, through time, with familiar `gdb`-like operations.

Categories and Subject Descriptors D.2.5 [*SOFTWARE ENGINEERING*]: Testing and Debugging—Tracing

General Terms Human Factors, Languages, Performance

Keywords cross-layer debugging, dynamic analysis, time-travel debugging

1. Introduction

Debuggers, as a tool in software development, have changed remarkably little over the decades in which they have been used. Most languages and runtime environments in use today have them; some may offer scriptable interfaces [1, 3] to automate debugging during regression tests or may allow developers to walk execution back in time in search of the source of a bug [12, 16]. These are all, however, relatively simple variants on a core idea: like a microscope or magnifying glass, the debugger is a tool that lets a developer take a *detailed* look at program state at a *single point* in time.

In this brief position paper, we argue that this established view of program debugging is inherently limiting. What if, as an alternative, the debugger had access to *all* states of an executing program *throughout* its execution? We propose a system in which the debugger is "unstuck in time",[1] allowing traditional debugging operations, such as inserting a breakpoint, to be evaluated throughout the entirety of a program's execution.

Allowing analysis on the entirety of execution invites debuggers to be thought of in an entirely different way. Developers are invited to ask questions about the overall dynamic behavior of a software system: what are the frequent values passed to this function? What path through the OS's I/O subsystem are traveled by requests from a specific application? Moreover, the system facilitates questions that begin broadly, for instance by examining all stacks that have existed during the invocation of some function, and then progressively refine to a narrower scope, either considering optimizations for the common case or understanding unusual outliers.

The end goal of our system is to build a debugger that is comprehensive through space and time: we want to be able to explore the behavior of software at any layer of the system (OS, application runtime, library or application code) and at all points during execution. Constructing a system that meets these goals has proven to be a challenging task. The work in this paper extends *Tralfamadore* [14, 15], an execution recording and offline dynamic analysis engine that has been developed in our research lab. We describe a debugger interface that we have built atop Tralfamadore, called *TDB*, that provides a `gdb`-like interface to execution traces. We explore how traditional debugging commands are different when run over an entire execution and describe our prototype implementation, integrated with a browser-based IDE.

Implementing a debugger over an execution trace presents a number of interesting technical challenges. For instance, as the trace stream records events such as writes to memory as they happen, answering questions such as, "What value is at address x at time y?" may involve a complex and expensive scan through the entire execution history. We describe extensions to Tralfamadore that have been implemented to address issues such as these and how they are used to build a number of useful debugging tools.

The remainder of this paper provides a snapshot of our current TDB prototype and is intended to solicit discussion and feedback on the tool. We begin by comparing TDB to UNIX's `gdb` and exploring how gdb's operations change in our system. We then describe some of the interesting technical challenges that we faced in building the system. The paper ends with a discussion of future work and the challenges that we anticipate.

2. TDB

Debugging through time requires both a debugger and a methodology that are conceptually different from conventional tools. We address the former in this section by presenting an overview of TDB and its functionality. Prior to this discussion, however, it is perhaps helpful to meditate on common activities in a traditional debugger and how those actions map to TDB. We describe a number of GDB–TDB equivalences in Table 1.

Traditional debugging relies on earmarking a few potentially critical actions or events and observing the effects they have on the entire system. On the other hand, debugging in time allows one to watch for particular effects and track back to the causes of said effects.

Consider running a program in an ordinary debugger and setting a breakpoint on a commonly called function such as `malloc`. Each

[1] This term, and the name of our analysis system, are borrowed from Kurt Vonnegut's novel, *Slaughterhouse Five*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RESoLVE '12 March 3, 2012, London, UK.
Copyright © 2011 ACM . . . $10.00

time the breakpoint is hit, execution is suspended and control returns to the debugger, where the developer can perform such tasks as studying the call stack at that moment and inspecting what argument was passed to `malloc`. In TDB, however, rather than being presented with data from n distinct breakpoint invocations, they are aggregated into a single compound data structure. In our example, all backtrace stacks would merge into a tree, with `malloc` at the root and different code paths that led to that function call fanning out from that point.[2] An example of a so-called *multi-backtrace* may be seen in Figure 1(b). Additionally, since we are considering all calls to `malloc` simultaneously, the values of the argument to the function aggregate into a histogram of all values observed over the course of the program's execution.

2.1 Workflow

TDB runs entirely within the developer's Web browser. While there already exist frontends to Tralfamadore implemented as IDE plugins, we felt this was a more appropriate delivery vehicle given our long-term goal of Tralfamadore acting as a distributed record and replay mechanism, as we explain in Section 3.3.

The TDB interface can be seen in Figure 1, where debugging actions will involve interacting with histograms, trees, graphs, and other visual representations of trace data. A TDB debugging session typically involves iteratively issuing queries to the debugger, analyzing the output, and refining those queries to filter out progressively more of the execution trace. Filtering allows interesting points of execution to be selected for subsequent comparative analysis. The programmer may want to filter based on a predicate relating to the most frequent, least frequent, highest, lowest, and/or unusual values of an argument to a function and see the state of execution when these values are encountered. For example, Figure 1(a), a *varvals* query on the `__alloc_skb` function in the Linux kernel, demonstrates that sometimes network buffers of just one byte are allocated. Using this predicate, one can run a *mbt (multi-backtrace)* query to determine that this situation arises in a very specific code path as shown in Figure 1(c). In a conventional debugger, a breakpoint set on a line of code in a library function may be hit too often to be useful and conditional breakpoints can't be used unless one knows the interesting (e.g. error) conditions *a priori*. In TDB, conditions on breakpoints can be refined as necessary without having to worry about reproducing the behaviour in a subsequent debugging session, because the same trace is used for each analysis. For example, the contents of 1(c) is actually a subset of 1(b), rather than being a separate run.

2.2 Desired debugger features

In the current TDB prototype, a user will often perform a series of actions like calling the TDB breakpoint command (resulting in a set of timestamps) and then filtering out uninteresting results. Allowing common refinement operations such as `break line-number if condition` could be beneficial both in terms of user-friendliness and efficiency of backend operations.

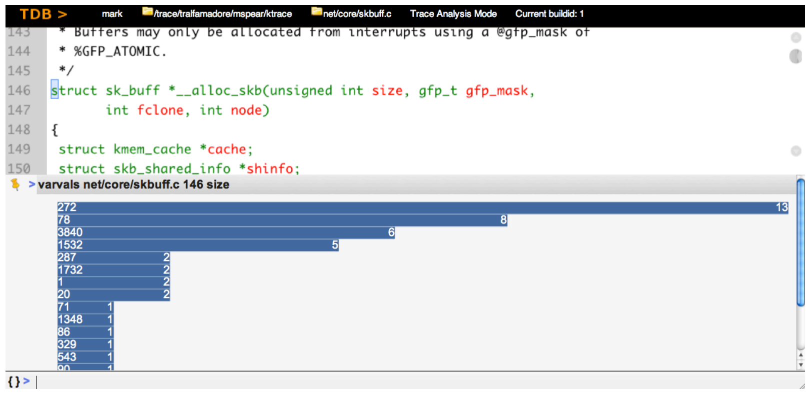
Additional temporal filtering techniques would be useful (e.g., for detecting if a piece of memory has been accessed after a free but before it has been reallocated) but have not been implemented yet. Other types of filtering might be to examine only breakpoint hits whose back trace does or does not include a particular function or to examine only breakpoint hits where a particular variable has or does not have a specific value. It is clearly possible to imagine arbitrarily complex predicates involving function call containment and variable values; how exactly to usably, intuitively implement such predicates remains open for future investigation.

Exploiting debugging information from multiple layers in the system (kernel, application, etc.) could allow for easier debugging of large systems and the ability to filter out uninteresting parts of the whole-system execution trace. Debugging information is also necessary to access the values of arbitrary variables which may, over their lifetime, move between locations on the stack or in global

GDB	TDB-equivalent
Break and Watch	
breakpoint	Rather than stopping and resuming execution as in GDB, breakpoints in TDB are sets of timestamps for points of execution which match the breakpoint target
Line Execution	
step, next	Not supported. In the TDB model of breakpoints, stepping would occur for multiple points of execution simultaneously. As execution can diverge along different code paths, the results could be difficult to interpret. We discuss this case more in Section 4.1.
until	Not yet implemented, but perhaps this makes more sense than single stepping if the developer has reason to believe the current breakpoint timestamps will converge at a given piece of code.
where	Map set of timestamps back to source:line info
Stack	
backtrace	*mbt* (multi-backtrace): Show a tree of backtraces. In the trivial case where the user has refined the breakpoint context to a single timestamp, an mbt query is equivalent to a backtrace. In the general case mbt shows a graph of the backtraces for all selected timestamps merged into a visualization which shows the relative frequency of call graphs that result in the selected/refined set of timestamps
info args	*varvals*: Show a histogram of variable values for a function argument over (the selected subset of) the trace
Source Code	
list	*list*: Bring up the relevant source code in the IDE with shortcuts for generating various queries (e.g., breakpoints on any line, varvals for function arguments, ...)
Examine Variables	
print	*print* displays values of a variable at the selected timestamps. This is still in progress as the location of the desired variable can be different (e.g., in a register registers vs on the stack) at each individual timestamp. Future work in multi-layer debugging will be to use specified or inferred datatypes to display formatted structures, rather than raw data values
Start and Stop	
run, kill	Preliminary work has been done on buildbot-style automation for trace collection of a VM running a predefined workload

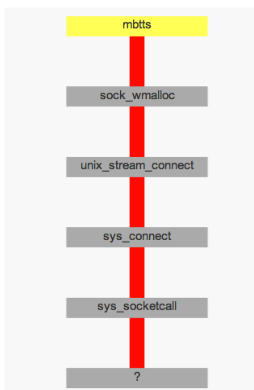
Table 1. GDB-TDB concept mapping.

[2] More precisely, this would form a lattice-like DAG with the bottommost nodes corresponding to the program's entry point. In the case of a kernel, by virtue of being event-driven, the entry point need not be unique.



(a) The TDB interface with the results of a *varvals* query, showing the distribution of values for the *size* argument of `__alloc_skb`.

(b) A *multi-backtrace (mbt)* query for a specified line of code reveals the many contexts in which it was encountered. Note that the full graph is much larger than what we can reproduce in this paper.



(c) A *mbt* query for the same line of code as Figure 1(b), but restricted to timestamps where the `__alloc_skb` function is called with $size = 1$. This situation arises via a unique call stack in the analyzed trace.

Figure 1. The TDB interface and the results of queries.

memory and registers, as well as to decode complex structures and evaluate arbitrary expressions for display to the user.

Another tool provided by any traditional debugger is the *watch point*, which halts execution whenever a particular location in memory is modified (unlike a breakpoint which halts execution whenever a particular location is *executed*). Watch points are often used if the programmer believes a variable's value is being corrupted, perhaps by a stray pointer or a buffer overrun, and wants to find out where this corruption is occurring. In TDB, the concept of a watch point can have two distinct interpretations. First, a watch point can be an entry point into a debugging session, similar to a breakpoint: a programmer can simply query a list of all places where a variable was modified. Second, a watch point can be used in combination with an already-executed breakpoint, to respond to such requests as as, "Show me, for each of these breakpoint hits, where this variable got its value." This latter form involves shifting the set of timestamps representing the current debugging session backwards in time; how to represent such changing debugging contexts remains an open question. How to represent situations where locality of source code is broken is another open question: given a breakpoint hit, for example, all timestamps in the resulting set occur on the same line of source; a watch point, on the other hand, may, and often will, be hit on many different lines of source. Similarly, a watch point used as entry into debugging to establish an initial context may also result in hits on different lines of source. Additional investigation is needed to determine how to display debugging contexts containing such disparate points to the programmer so as to allow him or her to rapidly drill down to interesting regions of execution while filtering out extraneous hits.

Incorporating such additional debugging features should be possible because the Tralfamadore execution trace contains complete execution information, but the practicality and efficiency of such queries have yet to be determined.

3. Tralfamadore

Tralfamadore is a framework for performing post-hoc dynamic analysis of a whole system, including both the operating system

kernel and userspace applications. In Tralfamadore, the system under test is first executed in a virtual machine which is instrumented to record enough information to subsequently replay the execution. In particular, we make records of all non-deterministic input to the machine, such as network traffic and hardware interrupts. This so-called deterministic replay log is sufficient to, when combined with an initial memory snapshot, reconstruct with perfect fidelity the entire execution. On replay, the log is expanded to form a *trace* containing every CPU instruction executed during the test.[3]

Once the trace is present on disk, Tralfamadore provides a number of different modules that use the trace for various purposes. As the trace is very large, indices can be constructed allowing the system to rapidly find interesting parts of the trace, something that would be impractical with a linear scan. For example, an index can be created showing each time a function is called; this index can then be used to rapidly "hone in" on points of interest while debugging or performing more complex but more targeted analysis.

Complex analyses are built out of several modules, called *operators*, that compose in a pipeline structure to incrementally add semantic meaning to the trace data. For example, one common piece of information that analyses might require is knowing about context switches. Since the trace sees the whole system's execution, context switches could be between the user-mode application and the kernel (due to a system call or hardware interrupt), between two different non–interrupt handler kernel contexts (e.g., kernel threads or system calls) via a stack switch in the Linux `schedule` function, or between a kernel thread or system call and an interrupt handler upon receipt of a hardware interrupt. The analysis begins with an operator that parses the trace file and emits *annotations*, individual atomic fragments of information that flow through the pipeline, for each basic block or hardware interrupt. These annotations can be examined by the other operators; those operators can also inject their own annotations of other types for use by yet further operators. For example, the *context operator* watches for hardware interrupts and invocations of the Linux context-switching function and emits *context annotations*, an abstract representation of context switches. These context annotations provide a unique context ID for each context no matter how it was reached, allowing further operators to track contexts and context switches easily without having to understand the details of the context switching mechanism.

3.1 Memory Reconstruction

One of the most important Tralfamadore modules for building a debugger is the *memory reconstruction module*. This module provides the ability to efficiently determine the value stored at a particular location in memory at a particular time. This might then be displayed to the programmer who wishes to know the value of a variable, for example.

Because the debugger must be able to obtain the value at any arbitrary point in the execution timeline, it is not sufficient to simply construct an image of memory ahead of time and then examine it. Instead, we determine the value in memory at a particular location by finding the point in time at which that location was most recently written to. To avoid a linear scan of the entire trace, it is necessary to use an index to more quickly find these points in time. The memory reconstruction module includes a tool that performs a single linear scan of the trace and generates this index, which can then be used very quickly. To construct the index, the tool first breaks the trace into disjoint temporal subintervals, called *epochs*, such that each epoch contains approximately 50,000 memory modifications. The tool writes an index record for each epoch containing the interval covered by the record and a run-length-encoded bitmap of which

[3] In theory, a temporal subinterval of the replay log could be expanded to save time and disk space; in practice, we always expand the entire log.

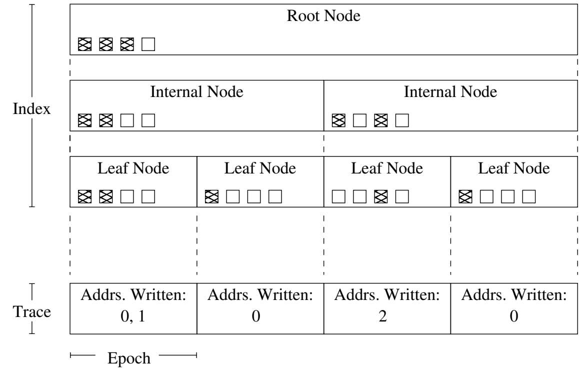


Figure 2. Memory index for a four-epoch-long trace of a hypothetical computer with four memory addresses

bytes in memory were modified within the epoch. Once all of these records are written, the tool then generates additional records covering larger intervals and forming a binary tree structure, the root node of which represents the set of bytes of memory written to over the entire trace.

To reconstruct the value in memory at a particular address and time, the reconstruction module uses the binary tree to find the epoch containing the last write to the address prior to the requested time.[4] Once the required memory write is localized to a single epoch, a linear scan of the epoch is performed in the trace file to find the actual write operation. The cost of this operation is thus the cost to navigate the tree, whose height is logarithmic in the number of epochs (and consequently in the number of writes in the trace), plus the cost to perform a linear scan over one or, in the worst case, two epochs of trace, each of which are of approximately constant size regardless of the length of the complete trace.

3.2 Address Translation

x86-based operating systems available today virtualize applications' views of the address space. The CPU does this transparently via a combination of two mechanisms: the page tables and the translation lookaside buffer. *Page tables* are data structures stored in memory that specify the translation for each virtual address. Because walking the page tables requires a number of reads from memory, it is quite slow. To avoid slowing every memory access by an application, translations are cached in the *translation lookaside buffer*, or TLB; subsequent accesses to the same virtual page will use the cached translation.

Naturally, one of the tasks a debugger with access to physical memory will need to perform is translating between virtual and physical addresses. Notably, unlike the regular CPU instruction and data caches, the TLB does not participate in cache coherence protocols and can become out of date with respect to the actual page tables in memory, even if the page tables are modified by *the same* CPU; instead, an explicit flush instruction is used to invalidate TLB entries. We would like our debugger to reflect this behaviour and use translations that are always the same as those the CPU would have used, even if an operating system fails to promptly flush the TLB (due to a bug or situation in which staleness is irrelevant).

To satisfy these needs, Tralfamadore includes an address translation module. This module accepts a virtual address and a point in time and returns the corresponding physical address at the speci-

[4] If the address is written to during the epoch containing the requested time, it may be that the write actually occurred after the requested time; in this case, a second run is used to find the next earlier write.

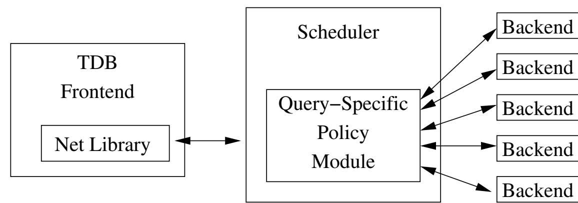


Figure 3. Parallel architecture hypothetically applied to TDB

fied time (or an error if the page was marked as absent in the page tables). For fidelity, we store information about the CPU's TLB activity alongside the trace while expanding the replay log. When translating, Tralfamadore first reconstructs the TLB state at the requested time to check whether the requested address appears; if not, Tralfamadore uses the memory reconstruction module described in Subsection 3.1 to reconstruct and walk the page tables. As address lookups are quite common activities, for increased performance, a preprocessing pass is executed when preparing to analyze a trace which indexes the timestamps of TLB-related events and enables rapid lookup.

3.3 Parallel Execution

In addition to the use of indices, another way to speed up execution of operations is to parallelize them. Tralfamadore's architecture is designed to support parallelization of queries both across processors in a machine and across multiple machines on a network. To parallelize an analysis task, it is divided into *frontend*, *scheduling*, and *backend* parts, which communicate over a network connection and will typically run on separate computers.

The frontend component is responsible for decomposing the analysis task into individual queries, processing the results of the queries, maintaining any needed state from query to query, and interacting with the user; in TDB, for example, the frontend part would[5] consist of the JavaScript running in the user's browser plus the Web application components running in the Web server.

The backend component is stateless for most requests and is responsible for accepting a request, processing it, and returning the results. The backend part is not tied to the frontend; thus, TDB and other frontends share a common backend. The backend is implemented as a single network dæmon with the requests built in as loosely-coupled modules. Many such modules do their work by constructing a pipeline of operators and running the pipeline against the trace, but some request handler modules do something more complex such as invoking the memory reconstruction module. Because the backend component is generally stateless, a typical installation will have multiple backends running on different computers for load balancing purposes.

The scheduling component[6] is responsible for accepting a request from a frontend, possibly dividing it into multiple smaller requests, and then issuing the request or requests to one or more backends.[7] The scheduler then collects the responses from the backends and returns them to the frontend, reorganizing them if necessary. The scheduler is where parallelism policies are defined for each type of request, and these policies can be constructed as needed. A scheduler can accept a request with no specific policy, in which case the request will be forwarded verbatim to a single backend; even this will grant some speedup as multiple simultaneous requests can be directed to different backends. If additional performance gains are needed, a policy module can be written which knows how to parallelize a particular type of request.

TDB's "breakpoint" command provides an example of a simple but common parallelization policy. The initial breakpoint request is issued to find all hits on the target address within a particular time interval. Parallelizing this request is very easy: the parallelization policy would simply divide the requested time interval into a collection of smaller time intervals, then issue a breakpoint request over each smaller time interval to a backend. Each backend will return the breakpoint hits contained in its time interval; the policy module need only interleave these hits and return them to the frontend.

The "multi-backtrace" command is also easy to parallelize, though not in the same manner as the breakpoint command. A multi-backtrace request carries a list of points in time for which backtraces are desired; parallelizing it is as simple as distributing those points to different backends. Some care should be taken, however, since it may be possible to improve performance by sending points that are close together temporally to the same backend.

Generation of context IDs is also fairly easy to parallelize, though it does require more work than breakpoint and multi-backtrace commands. This is because once the trace is divided into subintervals of time, a particular scan executed on a backend may encounter a particular context for the first time even though that is not the first occurrence of the context in the trace as a whole (the prior occurrences may lie in other backends' scan intervals). Instead, the backends would generate pseudo-context records which would be stitched together by the scheduler policy module into coherent, globally-uniquely identified context records.

3.4 Resolving debug symbols

Like all debuggers, TDB requires a mechanism by which semantic components of the original source code, such as variable names, function argument locations, and type information, are saved by the compiler and stored for later access. On Linux and other systems that support executing ELF binaries, the most frequently-used debugging format for C and C++ programs is the DWARF specification. As part of Tralfamadore, we have implemented a debug format-agnostic dæmon that allows TDB and other frontends to the analysis system to reconstruct human-readable components of the original source code.

While DWARF is a language-neutral standard, it is designed to emit debug information for languages with lexical scoping, such as C, C++, or Fortran. As a result, the primary DWARF structure is a tagged tree of so-called debugging information entries (DIEs), which allows information for lexical structures such as code blocks, functions, and object files to be composed within one another in a manner analogous to the original program source. It also encodes separate information such as line numbers and typedef information, and includes a stack machine-like interpreter for resolving register offsets at analysis time.

When a TDB session is initiated by the user, Tralfamadore opens a socket to the debugger dæmon, which forks and parses the ELF headers of the chosen kernel that contains the debug data. Over the same socket, the Tralfamadore backend issues CLRF-delimited queries to the dæmon. A common use case for this subsystem is to find correspondences between source line numbers and EIPs; for example, setting breakpoints requires finding the latter given the former, and visualizing multi-backtraces within TDB requires the former given the latter.

While support for resolving preprocessor macros is present in the DWARF standard, the difference between unprocessed source

[5] The current version of TDB was developed before this architecture was designed and thus uses its own private networking architecture and not the standard Tralfamadore parallel architecture.

[6] This component has not yet been implemented; we describe later in this section how it need only apply to Tralfamadore installations running with multiple backends.

[7] The protocol used on both sides of the scheduler is identical, so the scheduler can be omitted in a system with one backend.

files intended for human consumption and the rewritten ones intended for compilation introduces a significant semantic gap in a macro-heavy codebase such as the Linux kernel. We are therefore unable to reason about function-like macros with the same degree of precision as first-class C functions. Emitting more detailed macro debug information would be a fruitful activity, though it would unfortunately require compiler modification.

4. Future Work

4.1 Stepping

At present, TDB is only capable of analyzing the state of the system at a single point in code, where that point is potentially hit multiple times throughout the course of the trace. An interesting enhancement would be to allow the equivalent to a traditional debugger's "single step" command, which would advance all the collected timestamps by one source-language statement. If a conditional or indirect function call appeared in source, this might result in a split in control flow, with some executions following one path and some following another.

Such a command would result in an explosion of data as the number of stepped-over branches increases; an open question is how to best display this information. Should all the execution contexts be displayed, and each single-step advance all the contexts by one step independently? In the case of a loop, should only those contexts remaining in the loop advance, in the expectation that the contexts will eventually rejoin? Is it more useful to duplicate the debugger interface and allow the programmer to drive each branch of execution independently, or perhaps discard some branches if they prove uninteresting? Perhaps different options are more useful in different situations, and the programmer should be allowed to choose how a branch will be handled when it appears.

4.2 Automatic Case Comparison

Given a collection of traces covering similar executions of similar code paths (perhaps produced by running a collection of unit tests), where one trace contains an error of some sort and the others do not, can a tool automatically do at least some of the work of tracking down the differences between the two traces, saving a human programmer time and effort? Tralfamadore provides a uniquely powerful environment for observing both control flow and data flow as a program runs, safe from any possible nondeterminism.

One possible approach is to generate a backtrace from the point of failure in the failing run, then try to match subtrees in the backtrace with particular points in the execution of the successful runs. A failed match may indicate a point of divergence between a successful run and a failed run. Points of divergence could be ruled out as the cause of failure by comparing them to other successful runs; divergences that also appear between successful runs are not uniquely responsible for the failure. Alternatively, the system could automatically scan all the variables in the vicinity of the failure (perhaps the last few functions in the backtrace), find corresponding points in the successful runs, and compare the values of all global variables and locals within the top few functions; values that are common to all successful runs but that differ in the failing run may be responsible for the failure. Clearly such a tool will not automatically fix bugs, as it at best answers the "how" but not the "why" question, but it may significantly reduce the time spent by a programmer on narrowing down the cause of the bug.

4.3 Userspace Applications

Currently, TDB does not support debugging userspace applications, only the kernel. While Tralfamadore successfully records execution in both CPU modes and our system is able to replay a deterministic replay log comprising all layers of the software stack, debugging userspace applications is actually significantly harder for a number of reasons. First, multiple independent userspace processes invariably exist on any given system; even an ostensibly single-purpose system always has a number of control and helper dæmons running in the background. Second, userspace applications are subject to paging; the particular piece of data or code the programmer wishes to examine at some point in time *may not exist in memory*.

Determining where a page of code or data is actually located becomes a nontrivial exercise: in the simplest case, the page is present in physical memory and mapped into the process's virtual address space. Alternatively, it could have been brought into physical memory by another process but not yet mapped into the debuggee's virtual address space. In such a case, the in-memory copy can be discovered by introspecting on kernel data structures. In the event the data has been evicted from physical memory, it exists only in swap, in the executable binary image, in a shared object, or in a memory-mapped file; the data can then be retrieved by introspecting on kernel data structures to determine the source of the data, then parsing the partition table and filesystem structures from a reconstructed disk image from the appropriate point in time.

4.4 Debugging SMP Targets

Modern processors have reached a speed wall in which clock frequencies cannot be increased any further. Hardware designers' response to this has been to provide parallelism in the form of hyperthreading, multicore CPUs, and multiple processor sockets on motherboards. While hardware parallelism used to be the purview of servers, it is now becoming ubiquitous in desktops and even laptops. For Tralfamadore to continue to be useful into the future, it is clearly essential that it be able to debug operating systems running on parallel hardware, particularly since hardware parallelism often introduces hard-to-find bugs such as race conditions.

Some work has already been done in deterministic replay of parallel virtual guests, as will be noted in Section 5.4. Many of these replay mechanisms work by interleaving the multiple CPUs' instruction streams to construct a serialized execution whose externally-visible effects are equal to those observed during the original recording. Given the existence of such a replay mechanism, one could very easily construct an execution trace during the replay of the serialized instruction stream; assuming the addition of small amounts of data to the stream (e.g., on which CPU each instruction executed), the existing architecture would be capable of analyzing such traces.

One open question in this area is how best to display the interleaved execution to the user. Although many of the replay mechanisms will often generate much coarser-grained interleavings than the original execution, displaying the interleaving is still unnatural for a programmer, especially if the two CPUs are executing unrelated code. In fact, different displays may be appropriate in different sitations; sometimes a programmer will be interested in just one CPU executing one code path (and may want to see other CPUs' influences on memory as akin to devices' DMA transfers), while at other times a programmer may be interested in the precise interleaving that leads to a parallelism-related bug.

4.5 Debugging High-Level Languages

Because Tralfamadore records so much detail about system execution, it may be useful for debugging performance problems that manifest themselves in high-level languages with managed runtimes. Performance problems in high-level, managed languages can come from many sources, from the operating system to the native userspace environment to the managed runtime environment to the program itself. Given a particular operation that performs poorly, it may be difficult to determine which of these layers is responsible for the slowdown; furthermore, the slowdown may be caused not by

a single layer, but by the interaction of different layers. For example, one could easily conceive of an operating system paging algorithm that works well for ordinary native applications and a managed runtime garbage collector that follows sound principles and performs well on many platforms, but a deployment of the garbage collector on top of the paging algorithm may perform poorly due to neither algorithm having the "complete picture" of what's going on in memory. Tralfamadore could be extended to support the ability to separate the managed runtime environment from the rest of the trace and display, in a semantically-meaningful way, the state of the high-level-language program running therein. This would enable it to be used to debug such cross-layer issues by seamlessly following an operation from a program, through its managed framework, down into the lowest layers of the operating system, and back up again.

4.6 Parallel Universes

While TDB currently considers all points in time through an execution, it is limited to this particular unfolding of events. As the fidelity of recording is sufficient to instantiate a complete, running virtual machine at any point during recording, it might be interesting to consider allowing the system to fork and run in multiple directions—especially during testing. In this manner, the dynamic analysis techniques used by TDB might take a step in the direction of more static techniques where a more complete set of program states can be considered. We are interested in exploring the idea that a developer could assert some specific condition in TDB, for instance "Find a situation where variable x is modified but lock y is not held." In this case, the analysis engine might realize that this conjunction never occurred in the trace, and rather than returning failure, would attempt to fork execution from a state that did occur, and synthesize execution in a manner that produced the desired case.

4.7 Visualization

Although the majority of this work describes TDB, a debugger based on Tralfamadore, this is not the only tool we intend to build on Tralfamadore. Specifically, we would also like to investigate "program understanding" tools which would help a new programmer get up to speed quickly in an unfamiliar codebase. While some of the tools making up TDB would no doubt be useful in such a situation, such as histograms of variable values (to answer such questions as "what is the *normal* way in which this function is used?") and multi-backtraces (to answer such questions as "who *usually* calls this function?"), specific purpose-built tools are likely to be more valuable here. How to visualize a large amount of execution trace remains an open question, and we may consider working with software engineering and visualization researchers on these applications.

5. Related Work

5.1 Dynamic Analysis Frameworks

Our work builds upon the Tralfamadore project. While our ultimate goals are in line with those of earlier work [15], here we more rigorously define mappings between the functionality of traditional debuggers and TDB and are not merely interested in the architecture of the Tralfamadore backend.

There exist numerous binary translation-based systems for dynamic analysis of running programs [17, 21]. These tools differ from ours in the sense that they do not record a trace-based history of execution but rather perform all analysis at runtime. This is done typically by embedding a just-in-time compiler to add lightweight runtime callbacks [17] or heavier memory-centric analyses [2, 21]. These particular tools, however, are limited to userspace applications and therefore are unable to perform full-system analyses as we propose in this work. More serious, however, is the runtime overhead that such tools result in: the literature cites slowdowns of two to three orders of magnitude, enough to potentially affect the outcomes of non-deterministic events such as data races. Lastly, these systems, while extensible by virtue of their plugin support, are not intended for interactive use by a developer and instead typically output relevant data at the end of the instrumented program's execution.

5.2 Whole-System Analysis

Dynamic analysis of an entire running system is a tantalizing-enough notion that numerous such systems have been built, often as extensions of userspace tools [5]. Since the lowest levels of the software stack must be instrumented, typical approaches either emulate the processor [6] or run the analysis tool as a component in a virtual machine monitor [7, 8]. Because of the deluge of data that results from whole-system logging, these tools share Tralfamadore's post-hoc analysis modus operandi. As with their userspace cousins, these frameworks do not offer an interactive front-end as a core component of the system and tend towards being designed to pinpoint specific classes of problems, such as intrusion detection [9] or shared memory dependencies [20].

5.3 Time-Travelling and Omniscient Debugging

Many major open-source debuggers feature backwards-stepping functionality [1, 3]. However, the authors state that these features are not intended for backwards debugging so much as rolling back near-immediate state in situations where the user has stepped too many instructions. Additionally, backwards-stepping is not considered a first-class feature of the software; in the case of `gdb`, a specific `record` command must be explicitly invoked prior to execution. Other work has taken place building so-called omnicient debuggers [16] that feature arbitrary temporal traversal capabilities; however, work in this space tends toward the safer world of managed languages rather than programs running in native execution environments.

5.4 Deterministic Record and Replay

TDB uses deterministic virtual machine (VM) record and replay [9] to capture the execution of a complete system including the operating system kernel. The log is then replayed to generate a detailed trace. By decoupling trace generation from execution, it is possible to capture the execution of a single processor VM with an overhead often below 10% [26].

Recording the execution of multi-processor systems is substantially more expensive because the outcomes of memory races must be recorded. SMP-Revirt [10] uses shadow page tables to capture the ordering between shared memory accesses but the overhead can be prohibitive for workloads that exhibit high-levels of sharing. This technique is also prone to false sharing due to page size granularity.

Fortunately, recording multi-processor execution is an active area of research. Work such as PRES [22] and ODR [4] propose to use a weaker form of recording where the order of memory races is not recorded. This approach turns replay into a search problem and may fail to reproduce the original execution but the fidelity might be sufficient for a specific purpose such as reproducing a failure. DoublePlay [24] explores the insight that data races are relatively infrequent and uses speculative execution to predict that a serialized version of a multi-threaded execution will produce the same final state. If the assumption is true, the serialized version represents a valid recording; otherwise, execution is rolled back.

Record and replay is also an active area of research in the architecture community. Earlier work such as FDR [25] and BugNet [19]

relied on directory-based cache-coherence protocols and sequential consistency. More recent work [11, 13, 18, 23] aims to support record and replay for snoop-based hardware and more relaxed memory consistency models such as total store order, making them potentially suitable to be implemented on modern commodity processors (e.g., x86).

All of these projects are complementary and would be highly beneficial to TDB and Tralfamadore as they could be used to efficiently decouple the generation of traces for multi-processor systems from the analysis of those traces.

6. Conclusion

Understanding program execution, whether it is with the goal of fixing a bug or becoming familiar with a new subsystem, is a challenging task for which conventional debuggers present a mediocre solution. In this paper we presented TDB, a IDE-based debugger that is a front end for the Tralfamadore execution analysis engine. TDB allows conventional debugging commands to be applied across the entirety of a software system's execution and allows developers to interact with a complete execution trace that spans multiple layers of software. While the work we have described is an early prototype, the system is capable of performing interesting new operations that are not possible with conventional debuggers. As we solve the list of challenges described in our related work, we believe that this will represent a novel and useful tool for software developers.

References

[1] Project archer. URL `http://sourceware.org/gdb/wiki/ProjectArcher`.

[2] C/C++ trace-based debugger based on chronicle and eclipse. http://code.google.com/p/chronomancer/.

[3] Using scripting and python to debug in LLDB. URL `http://lldb.llvm.org/scripting.html`.

[4] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: http://doi.acm.org/10.1145/1629575.1629594. URL `http://doi.acm.org/10.1145/1629575.1629594`.

[5] P. P. Bungale and C.-K. Luk. PinOS: A programmable framework for whole-system dynamic instrumentation. In *Virtual execution environments*, 2007. ISBN 978-1-59593-630-1.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, 2004. URL `http://portal.acm.org/citation.cfm?id=1251375.1251397`.

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

[8] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with crosscut. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 13–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-910-7. doi: http://doi.acm.org/10.1145/1735997.1736002.

[9] 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 *Operating Systems Design and Implementation*, 2002.

[10] G. W. Dunlap, D. G. Lucchetti, M. Fetterman, and P. M. Chen. Execution replay on multiprocessor virtual machines. In *International Conference on Virtual Execution Environments*, 2008.

[11] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *International Symposium on Computer Architecture*, 2008. URL `http://dx.doi.org/10.1109/ISCA.2008.26`.

[12] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. pages 1–15, 2005.

[13] D. Lee, M. Said, S. Narayanasamy, and Z. Yang. Offline symbolic analysis to infer total store order. IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), 2011.

[14] G. Lefebvre. *Composable and Reusable Whole-System Offline Dynamic Analysis*. PhD thesis, University of British Columbia, Nov. 2011.

[15] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: Unifying source code and execution experience (short paper). In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09.

[16] B. Lewis. Debugging backwards in time. AADEBUG, 2003.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, 2005.

[18] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory mu ltiprocessor execution efficiently. In *International Symposium on Computer Architecture*, 2008. URL `http://dx.doi.org/10.1109/ISCA.2008.36`.

[19] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *International Symposium on Computer Architecture*, 2005.

[20] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[21] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, 2007.

[22] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: http://doi.acm.org/10.1145/1629575.1629593. URL `http://doi.acm.org/10.1145/1629575.1629593`.

[23] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a chunk-based memory race recorder in modern CMPs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 576–585, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-798-1. doi: http://doi.acm.org/10.1145/1669112.1669183. URL `http://doi.acm.org/10.1145/1669112.1669183`.

[24] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 15–26, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: http://doi.acm.org/10.1145/1950365.1950370. URL `http://doi.acm.org/10.1145/1950365.1950370`.

[25] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture*, 2003.

[26] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Modeling, Benchmarking and Simulation*, 2007.