Tralfamadore: Memory Reconstruction, Declarative Dependency Resolution, and Parallelism

by

Christopher Charles David Head

B.Sc., University of British Columbia, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

November 2011

© Christopher Charles David Head, 2011
Abstract

Tralfamadore is a platform for debugging and analyzing whole software systems, from the operating system up. Tralfamadore employs a novel mechanism for analysis in which, rather than interrupting and examining the system under test as it runs, the system is instead run to completion untouched then examined post-hoc. The system under test is run in a virtual machine which records its execution; analysis and debugging tools are then applied to the recorded execution trace data. Tralfamadore thus permits travelling both forward and backward through time while debugging, debugging systems that would not normally be able to be halted (such as those that communicate with time-sensitive external systems), and accurately reproducing a view of a system’s execution even if that system has (maliciously or accidentally) corrupted its internal state. Tralfamadore also has a number of other potentially-interesting applications in aspect-oriented programming, forensic analysis, dynamic analysis, and customer-site bug reporting.
Preface

Tralfamadore was constructed collectively by a group of students and faculty in the Department of Computer Science. The trace recording component was based on Fabrice Bellard’s QEMU CPU emulator.

Of the mechanisms described in Chapter 2, I alone designed the format of the on-disk memory index, wrote the tool that constructs the index, and wrote the tool that queries the index. I also wrote the postprocessing tools described in Section 2.3.3 that use trace data to find CR3 loads and INVLPG instructions to construct indices and the tool that reorganizes and indexes the list of TLB loads, as well as designing the on-disk format of all of these data and index files. I modified the QEMU component of Tralfamadore to output the TLB loads to a separate file beside the trace data instead of inside the trace data file, but the code to capture and log TLB loads was written by others. I wrote the tool that uses memory reconstruction to walk the system under test’s pagetables to perform arbitrary virtual-to-physical address translation in the event that a relevant TLB entry is not available.

Of the mechanisms described in Chapter 3, I designed and implemented the concept and mechanism of declaring dependencies (both statically and dynamically), the concept and mechanism of tag-specific filters, the concept and mechanism of priority ordering between multiple equivalent operators, and the entire algorithm for automatic pipeline construction as described in Section 3.3. I did not design the concepts of operators, operator pipelines, annotations, or tags; these components were added to Tralfamadore by others. I also did not design or implement the example operators described, the context and invoke operators.

Of the mechanisms described in Chapter 4, I designed and implemented the
libtralf client library, the network protocol, and the worker-side daemon interface. I designed the architectural concept of a collection of workers with a transparent proxy doing job scheduling, but implementation of the proxy was not my work.\textsuperscript{1} I did not singlehandedly design the queries described as examples in Section 4.1 though I participated in design discussions for all three examples.

The original Tralfamadore system will be presented in Geoffrey Lefebvre’s University of British Columbia Ph.D. thesis\cite{5}.

The original Tralfamadore system was published in a short paper titled “Tralfamadore: unifying source code and execution experience” in the Proceedings of the 4th ACM European conference on Computer systems, EuroSys ’09, pages 199–204\cite{6}.

\textsuperscript{1}It was, in fact, never completed.
# Table of Contents

Abstract .............................................................. ii
Preface .............................................................. iii
Table of Contents .................................................... v
List of Figures ....................................................... vii
Acknowledgments .................................................... viii

1 Introduction ......................................................... 1
  1.1 Related Work .................................................. 3

2 Memory Reconstruction ........................................... 6
  2.1 The Index ....................................................... 7
  2.2 Reconstruction ................................................ 7
  2.3 Virtual and Physical Addresses ............................... 9
    2.3.1 Translating Addresses from Instructions ............... 10
    2.3.2 Translating Arbitrary Addresses at Arbitrary Times ... 11
    2.3.3 Implementation ........................................ 12
    2.3.4 Overhead ............................................. 13

3 Automatic Dependency Resolution .............................. 14
  3.1 Annotations and Operators .................................. 14
  3.2 Dependency Declaration .................................... 15
  3.3 Pipeline Construction ....................................... 18
4 Parallel Query Evaluation .................................................. 21
  4.1 Examples of Parallelizable Work ................................. 22
    4.1.1 Breakpoint Scanning ........................................... 22
    4.1.2 Execution Context Scanning and ID Generation .......... 23
    4.1.3 Backtrace Generation .......................................... 24

5 Conclusion ........................................................................ 25

6 Future Work .................................................................... 27
  6.1 Memory Index ............................................................. 27
  6.2 Automatic Dependency Resolution .............................. 28
  6.3 Parallel Query Evaluation .......................................... 28

Bibliography ........................................................................ 30
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Index search algorithm</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Memory index for a four-epoch-long trace of a hypothetical computer with four memory addresses</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Dependency declarations for the breakpoint operator</td>
<td>16</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Parallel query execution architecture</td>
<td>21</td>
</tr>
</tbody>
</table>
I am grateful to my supervisors, Andrew Warfield and Bill Aiello, without whose support and guidance I could not have reached this point, and the second reader for my thesis, Norm Hutchinson. I would also like to thank the many excellent teachers I studied under during my time as an undergraduate and graduate student at UBC and who between them guided me toward my academic career.

I would also like to extend my gratitude to the other students in the Networks, Systems, and Security lab. Many of them worked on Tralfamadore, and without them the project could never have succeeded; all of them provided a friendly and welcoming lab environment in which to work. I especially appreciate the assistance of Brendan Cully and Geoffrey Lefebvre, the original designers of Tralfamadore, for taking the time to answer questions and help me gain an understanding of the system.

Last but not least, thanks to my parents, David and Elaine, and my girlfriend, Wendy, for their support during my work on this degree program.
Chapter 1

Introduction

_You must not fall into the trap of rejecting a surgical technique because it is beyond the capabilities of the barber in his shop around the corner._ — Edsger W. Dijkstra (1975)

Debugging is hard, usually harder than initially developing a system, and debugging operating systems is harder still. Operating systems have to have very high performance to be generally accepted, must remain secure while exposing a massive attack surface to applications, must work in concurrent environments, and must communicate with a wide variety of often-quirky hardware. When something goes wrong, debugging tools for operating systems are often harder to configure and less powerful than those for userspace applications. Meanwhile, while among the hardest pieces of software to debug, operating systems are also exactly those pieces of software where reliability is most important!

One of the hardest problems in debugging an operating system is that, unlike a regular application where a debugger can be effectively invisible by running in a separate process, traditional operating system debuggers run as part of the operating system being debugged, as until recently there was no other choice. This unfortunately means that some classes of bugs in the operating system could damage the debugger’s code or data structures, or that the actions taken by the debugger to communicate with the programmer could influence the execution of the operating system’s code in more varied and unpredictable ways than might happen with
an application debugger.\footnote{For example, a debugger using a serial port to communicate would have trouble debugging a serial port driver.}

Tralfamadore takes advantage of the capabilities of virtual machines to improve the state of the art of operating system debugging. By running the operating system under test in a virtual machine with the debugger on the outside, one achieves an equivalent level of isolation to that achieved by out-of-process debuggers for ordinary applications.

Currently, Tralfamadore uses the QEMU system emulator\cite{1} for all its virtual machine needs. While QEMU’s performance is not as great as the more recent generation of virtual machines that use hardware virtualization capabilities, QEMU’s software-emulation model makes instrumentation easier. There is work currently in progress to port the initial recording portion of Tralfamadore to KVM\cite{4}, which will improve performance while the system under test is actually running in its regular environment.

Tralfamadore begins with the system under test running under QEMU (or, in future versions, KVM). The programmer orders the specially-modified version of QEMU to begin recording. QEMU first writes a snapshot of the virtual machine state to a file, then begins recording a log of all nondeterministic events such as interrupts, timer reads, keyboard and mouse input, and network traffic. The programmer exercises the system under test to expose the behaviour to be debugged, then halts recording. The resulting log file contains a record of the exact machine state at the beginning of the recorded time period plus a record of all nondeterministic events delivered during the operation; this information is sufficient to replay the original execution, with perfect fidelity, down to the level of individual CPU instructions.

The programmer then chooses an analysis tool to run against the recorded execution. Some analysis tools are written as integrated code inside QEMU, meaning they run inline during a playback of the recorded log file. Other tools may be larger or require random access to trace; these tools first require that trace be materialized, a process in which QEMU is run in replay mode and a trace file is written to disk containing the details of every single CPU instruction executed by the system under test (this process is slow and requires a large amount of storage, but does not
impede the performance of the system in its original environment because it can be
done during a replay). Still other tools may require auxiliary data from an index;
such indices are typically generated using a separate tool by either a replay run or
an examination of the trace file and then saved for use by the analysis tool.

Because Tralfamadore performs its analysis after the system under test has
already finished executing, rather than interactively against the system, as well
as achieving a high degree of isolation, the existence of the complete recorded
log allows debugging to occur in ways other than the traditional “searching and
stepping forward” model. For example, Tralfamadore can move both forward and
backward in time, search backwards to find root causes of observable effects, or
compare system state at different points in time.

1.1 Related Work

Bil Lewis’s Omniscient Debugger[7] is an earlier attempt at exactly the kind of
“backwards-and-forwards-in-time” debugging provided by Tralfamadore, but dif-
fers in a number of ways. First, ODB targets an application, rather than an oper-
ating system—the ODB described by Lewis targets Java applications, but he indi-
cates that other languages could be debugged similarly—and he does not indicate a
reasonable way of extending the capabilities to a whole operating system. Second,
ODB limits the amount of data that can be collected, requiring the programmer
to decide between discarding old data, reducing the coverage of the recording by
excluding certain functions, or recording a shorter period of time; Tralfamadore
has none of these restrictions and allows arbitrarily long executions to be recorded
at perfect fidelity, limited only by available hard disk space. Third, ODB claims
slowdowns of over $100 \times$ for some code patterns because it records individual pro-
gram activities; Tralfamadore, because it records only nondeterministic input to the
system from outside sources, is able to run most of the system completely unin-
strumented and hence at full speed.

Robert O’Callahan’s Chronicle[9] is another similar attempt. Chronicle, rather
than being implemented in the Java virtual machine for Java applications, is im-
plemented as a Valgrind tool for use with native C/C++ applications. Compared
to Tralfamadore, Chronicle still targets application debugging rather than virtual
machine debugging. Additionally, while Chronicle does take some basic steps to reduce the size of its recorded traces, it does not go as far as full deterministic replay the way Tralfamadore does. In fact, in chapter 12, for future work, O’Callahan states that “Another option would be to use a low-overhead recorder... to record at nearly normal speed, and then replay that execution offline under Chronicle to build a rich trace database.” This is virtually a perfect description of Tralfamadore’s operation.

King et. al.’s Time-Traveling Virtual Machine (TTVM)[3] is much closer to Tralfamadore in concept than either Lewis or O’Callahan’s work. TTVM uses something close to a virtual machine (User-Mode Linux does not quite satisfy Popek and Goldberg’s criteria[10], as it is not an unmodified guest kernel) to debug operating systems, rather than applications as Lewis and O’Callahan’s projects do. Also, TTVM uses full deterministic replay based on logging only those events that are truly nondeterministic, which Tralfamadore also does and which Lewis and O’Callahan’s projects do not do. However, TTVM, by targeting User-Mode Linux, does not actually debug a completely unmodified Linux kernel. While the device driver layer King et. al. provide does grant the ability to debug device drivers, a fully-unmodified kernel would clearly still be preferable. King et. al.’s work is appropriate for its era, being written in 2005 when full Popek-and-Goldberg-style virtualization was not generally available for X86 hardware. Today, with both Intel and AMD providing Popek-and-Goldberg-style virtualization in hardware, and with full virtualization becoming more and more commonly used commercially, a virtualization-based system like Tralfamadore has become practical.

Zhang and Gupta’s Whole Execution Traces (WETs)[11] approach the problem of storing (large) execution trace data, and one of the problems they address is efficiently examining memory modifications. However, their work views execution as a timeline of control flow activities and events, a representation which is suitable for profiling and analysis but not so useful for interactive debugging. While considering timelines is important, programmers are more familiar with the idea of a machine state that can be examined at a single point in time; my memory reconstruction tool for Tralfamadore provides this view in the form of efficient memory query resolution with random access in both time and memory address, whereas Zhang and Gupta’s work does not provide this random-access capability, instead
opting for efficiency when scanning and slicing the trace in a non-interactive way.

Query languages PQL[8] and PTQL[2] (Program Query Language and Program Trace Query Language, respectively) both provide high-level interfaces to analyzing program execution. However, both PQL and PTQL require the query to be known before the program under test is executed, as they use information about the query to reduce the amount of instrumentation they apply. Tralfamadore, in contrast, captures enough trace data to perform arbitrary queries after the system under test has executed. Further, their intended use is analysis of a program, not interactive debugging; they do not provide the power that Tralfamadore does to allow a programmer to interactively move between points in time while examining the system under test in as much detail as an ordinary debugger permits, a feature made possible by my memory reconstruction tool. Finally, while PQL and PTQL both provide high-level semantic querying of execution, their query-language-based model differs significantly from Tralfamadore’s operator-pipeline-based model, the latter inspiring my dependency resolver system; the dependency resolver system permits more flexible analysis of the system under test as one is not limited to using primitives defined in the syntax of the query language, but rather can write additional operators which simply plug into the existing framework.
Chapter 2

Memory Reconstruction

One of the most basic operations a programmer may request while debugging is for the debugger to display the value of a variable or expression. In a traditional debugger, this is fairly straightforward: the debugger simply reads the appropriate region of memory (or CPU register) from the suspended process. In Tralfamadore, however, an extra dimension is added to the problem: time. Rather than memory appearing as a one-dimensional array with a value for each address, memory is now a two-dimensional array where every address has a value at every point in time. Selecting a particular point in time slices the memory array, leaving an image of the memory as it appeared at that point in the system under test’s execution. The problem, then, is how to efficiently retrieve data from memory given both the address in memory and the point in time at which the data should be extracted.

The fastest possible query performance would come from simply materializing the two-dimensional array. Clearly, however, such an approach would require impossibly large amounts of storage, as it would essentially amount to taking a complete machine snapshot after every single CPU instruction. A better data structure is needed.
2.1 The Index

To enable efficient reconstruction of memory contents, an index file is created beside the main trace file. The index is structured as a full binary tree. Each leaf node represents a period of execution time covering approximately 50,000 memory write operations, which we refer to as an epoch; the node contains a compressed bitmap indicating which physical addresses in memory are touched by those writes, as well as pointers to the positions in the trace where the epoch begins and ends. Each non-leaf node contains a bitmap which is the union of its children’s bitmaps, and hence indicates which physical addresses are written to by the trace interval covered by both children; it also contains pointers indicating the total region of trace covered by its children, and hence by the node itself.

The index is initially constructed offline by scanning the trace from start to finish, writing out leaf nodes. Once the leaf nodes are generated, additional tree layers are generated up to the root node. Because index generation requires a scan of the trace, it can be slow; however, the resulting index tends to be quite small (e.g., a memory index for a 106 gibibyte trace taken while compiling a small Linux kernel is only 685 mebibytes in size, an overhead of just 0.6%).

2.2 Reconstruction

Reconstructing data in memory is conceptually done at byte granularity. A user of the reconstruction mechanism issues a query of the form, “Find me the value of the byte at address \( A \) in memory as it existed just before executing instruction \( I \).” As an image of memory is not available for arbitrary points in time, the reconstruction mechanism answers this query by finding which instruction stored the byte at address \( A \) and extracting the stored data from it; I shall refer to this instruction as \( W \). The instruction \( W \) is the one which writes to address \( A \) and for which no other instruction lies between \( W \) and \( I \) that also writes to \( A \), i.e., the last instruction before

---

1. I take a full binary tree to be a tree in which no node has exactly one child.
2. This statistic covers a trace file recording only operating-system execution; we often exclude user-level application code from the trace for space reasons.
3. For efficiency, a query of a larger block may be faster than a sequence of individual queries because the reconstruction code in practice may be able to take advantage of instructions that write more than one byte at a time, or of sequences of nearby instructions that write nearby addresses.
1. Check that the time interval covered by the node covers at least one instruction prior to $I$. If this is not the case, return failure: instructions occurring after $I$ cannot possibly influence a value seen at $I$.

2. Check that address $A$ is present in the node’s bitmap, meaning that $A$ was written to by some instruction covered by this node. If this is not the case, return failure: a time interval in which no writes occur to $A$ cannot possibly influence the value found at $A$.

3. If this node is a leaf, scan the corresponding interval of trace, looking for instructions that write to $A$. Keep the last such instruction that appears before $I$ and return success. Due to nodes covering epoch granularity, it is possible that $I$ may appear within the time interval covered by the leaf node; in this case, one must avoid considering instructions that appear at or after $I$, as they cannot influence the value seen at $I$. If no instruction is found that both writes to $A$ and appears before $I$, return failure.

4. If this node is a non-leaf, invoke the algorithm on its right child. Should the sub-invocation succeed, return success. Should the sub-invocation fail, invoke the algorithm on the node’s left child. Invoking on the right child first, then only on the left child if the right child fails, ensures that $W$ will not be set to an earlier instruction than it should be.

**Figure 2.1:** The index-search algorithm, executed for each node starting from the root.

**Figure 2.2:** Memory index for a four-epoch-long trace of a hypothetical computer with four memory addresses
I that writes to A. The reconstruction mechanism executes the algorithm shown in 2.1 in order to find W. A failure return from the algorithm indicates that W does not exist. This occurs if A was never written to between the start of the trace and I; the value at A is then extracted from a memory image snapshot taken at the start of the trace.

As an example, consider the memory index and trace shown in Figure 2.2, and assume the programmer asks for the value in address 0 sometime during the fourth epoch. Beginning at the root node, the algorithm verifies that, obviously, the root node covers some portion of the trace before the target instruction. The root node’s bitmap also contains a 1 in position 0, meaning the address has been written to during this interval. The algorithm proceeds to the right child of the root, the second node in the second level. The same process again determines that a temporal overlap exists and that a 1 is in position 0 of the bitmap, thus moving down to the fourth leaf node. The reconstructor now scans the fourth epoch in the trace file, looking for the write to address 0. If that write appears before the target instruction, the problem is solved: the write in epoch 4 gives the value seen at the target instruction. If the write appears after the target instruction, however, then the value present in memory at the point when the target instruction was executed would have been the value written in epoch 2. In this case, step 3 of the algorithm in Figure 2.1 will return failure. The internal node will then retry the algorithm against its left child, the third leaf node. That leaf node does not contain address 0 in its bitmap, so will omit scanning the trace (were it an internal node instead of a leaf node, it would also omit recursively examining its children). Because neither child of the second internal node succeeded, that node will itself return failure to the root node, which will then query its left child. This will lead to the second leaf node being queried and the proper value located.

2.3 Virtual and Physical Addresses

A complication arises from the fact that the architecture of the X86 CPU includes memory virtualization: running code does not see the physical addresses of the bytes of memory it accesses, but rather sees virtual addresses in an abstract address space. The operating system’s memory manager establishes a mapping from an ap-
application’s virtual address space to the physical addresses used to access memory; it does this by constructing an in-memory hierarchical page table which contains such mappings at the granularity of pages, which are typically 4 kibibytes in size. The operating system tells the CPU where the page table is located; as the application runs, the CPU walks through the page table to find how the virtual addresses used by the application correspond to physical addresses. To avoid the overhead of doing this walk repeatedly for every memory access, the CPU automatically caches a number of these mappings in its translation lookaside buffer, or TLB. Future accesses to the same page will use the value from the TLB, saving the memory accesses.

When designing the memory reconstruction mechanism, a choice had to be made between whether to represent memory addresses in virtual or physical form. Working with virtual addresses is easy because Tralfamadore records instruction side-effects using virtual addresses in the trace and because most queries will tend to be against virtual addresses. However, a naïve approach using virtual addresses suffers from the aliasing problem: it is possible for an operating system to construct page tables such that multiple virtual addresses map to a single physical address. Should this occur, writes to any one of the virtual addresses will naturally appear to modify the data residing at the other virtual addresses. Any mechanism built around virtual addresses therefore needs to handle this possibility, or else risk producing incorrect output. To avoid the aliasing problem, I use physical addresses.

2.3.1 Translating Addresses from Instructions

It would initially seem that one could, when working with virtual addresses, quite easily translate them to physical addresses simply by examining the contents of memory and manually walking through the page tables. Unfortunately, this suffers from a chicken-and-egg problem: to obtain the contents of the page tables themselves, one needs to reconstruct the contents of memory; recall, however, that Tralfamadore records memory writes in its trace file using only virtual addresses—including those operations that themselves write to the page tables!

Solving this problem relies on making an important observation about how a physical CPU works, a mechanism which is replicated in Tralfamadore’s CPU em-
ulator, QEMU: memory can only be accessed after its associated page table entry has been loaded into the translation lookaside buffer. Therefore, Tralfamadore can be modified to record, alongside the trace file itself, a log showing each time a page table entry is loaded into the TLB. By constructing an index over this log, it is possible to very quickly determine the virtual-to-physical mapping in effect at any point where an instruction reading or writing memory executes. This log is used to construct a mapping table during memory index generation that credits memory write operations to the proper physical addresses in the index bitmap, as well as during the last step of querying (trace scanning) to check whether or not a particular write operation hits the physical address requested.

2.3.2 Translating Arbitrary Addresses at Arbitrary Times

Using TLB loads is sufficient for index generation and for translating addresses of memory writes found in the trace during the final scanning step. However, it is not sufficient for general virtual-to-physical address translation: a user may request a virtual-to-physical translation at a point in time within the trace at which the corresponding page table entry had not been loaded into the TLB.

It is not appropriate to solve this problem by simply unconditionally walking the page tables, even though the chicken-and-egg problem has been solved at this point. The reason why this is not appropriate is "staleness": it is possible for the operating system to modify the page tables in memory without immediately clearing the corresponding page table entry in the TLB. Should this happen, the CPU will use the TLB entry, not the page tables in memory, to perform address translation; to maximize fidelity, the translation tool should do the same.

To solve these problems, a hybrid approach is needed: the TLB must be used if an entry is present for the target page, but the page tables must be used as a fallback if the TLB misses. This is identical to the semantics used by the CPU itself, and thus guarantees perfect fidelity. Thus, additional indices are created which allow the translation tool to rapidly determine the times at which TLB entries are invalidated (this can occur due to execution of the INVLPG instruction or due to a store to CPU control register CR3). When searching for a translation, the

---

While real CPUs may also invalidate TLB entries due to the finite size of the TLB, QEMU will never do this as its TLB is large enough to hold an entry for every virtual page.
translator first looks for a TLB load for the target virtual address preceding the target time, then checks for an invalidation following the TLB load. Should no TLB load be found or should it be invalidated before the target time, a page table walk is performed. Should a TLB load be found that is not invalidated until after the target time (or not at all), the TLB entry is used.

2.3.3 Implementation

TLB loads are written in chronological order to a raw log file while the system under test is running. In a postprocessing step, these records are transformed into an index file and a data file in a suitable format for fast querying. The index file is an array of index entries sorted by virtual page number; each entry in the index file contains pointers identifying a block of records in the data file corresponding to that virtual page. Each such block in the data file is sorted chronologically. To look up the TLB entry for a virtual page number at a particular time, the translator first performs a binary search on the index file to find the index entry for the virtual page number, then performs a binary search on the block of records in the data file to find the record closest to the requested time.

INVLP instructions are written, like all other instructions, to the main trace file by Tralfamadore. In a postprocessing step, the trace is scanned and these instructions are extracted and transformed into an index file and a data file formatted similarly to the files used for TLB loads.

Modifications to control register CR3 are also written to the main trace file. As the invalidation effects of these instructions are global, instead of affecting only a single TLB entry, the index/data pair described above is unnecessary. Instead, the postprocessing step simply extracts the times of the CR3 modifications into a file, keeping them in chronological order. Searching for the most recent CR3 modification prior to a particular target time consists of simply binary searching the data file.

\footnote{It is not practical to include them inline in the trace file because they effectively occur before the basic block that triggers them, yet their occurrence is unknown until the basic block is executed.}
2.3.4 Overhead

For the aforementioned 106 gibabyte trace, the raw TLB log file generated by Tralfamadore is 780 mebibytes. After postprocessing, the reordered data file is 585 mebibytes (smaller because it omits virtual addresses from each record), and the corresponding index file is 221 kibibytes. The data file containing INVLPG records is 624 kibibytes, and the corresponding index file is 9.7 kibibytes. The data file containing times of CR3 modifications is 948 kibibytes.
Chapter 3

Automatic Dependency Resolution

3.1 Annotations and Operators

Many queries in Tralfamadore are built as pipelines of simple operators through which a stream of data flows. Data is organized into discrete quantities, called annotations, which move through the pipeline. The first source of data in the pipeline is generally the trace file itself; the trace file provides an annotation for each basic block executed by the system under test, as well as special annotations for other events such as hardware interrupts, faults, and traps. Other operators can examine the annotations flowing past them and insert additional annotations into the stream at arbitrary locations. For example, two commonly-used operators in Tralfamadore are the context operator and the invoke operator.

The context operator uniquely identifies execution contexts in the target system. Each thread is considered a separate execution context; taking a hardware interrupt is also considered a new execution context because it is fundamentally unrelated to the code that was executing when the interrupt was delivered. The context operator examines annotations as they pass and, upon seeing either a hardware interrupt or a call to the Linux kernel’s context switching function, determines the new execution context being entered and generates an appropriate Context annotation.
The invoke operator identifies regions of execution time in the trace that are associated with invocations of a particular function. A function call is discovered by watching for execution of a basic block whose first instruction matches the address of the function, which is typically extracted from debug information provided by the compiler. The function’s corresponding return is discovered by watching for the execution of a `RET` instruction with the stack pointer, `ESP`, equal to the value it held on entry to the function.

Both of these pieces of information—context switches and matching of function calls and returns—are clearly useful for building higher-level semantic understanding of a system under test. Other operators could, and do, rely on these annotations.

Clearly, such a design produces dependencies between operators; a pipeline containing an operator that uses an annotation must also contain the operator that produces the annotation; furthermore, the consuming operator must be placed after the producing operator. It is inconvenient for a developer to be required to explicitly construct all the needed operators and insert them into the pipeline in the proper order. Manually constructing an operator pipeline may also cause maintenance headaches: if it is later discovered that an operator can be rewritten more efficiently by using another operator’s annotations which it did not formerly use, all pipelines containing the operator must be modified to contain its new dependency. For these reasons, an automatic resolution mechanism which allows each operator to declare which annotations it produces and consumes and which automatically constructs a pipeline satisfying the dependencies is desirable.

### 3.2 Dependency Declaration

The first part of automatic pipeline construction is accurate declaration of each operator’s dependencies. At startup, each operator registers itself with the system. The operator provides a data structure indicating all the `tags` (classes of annotations) that it produces as output and that it consumes as input. These pieces of information are used statically when constructing candidate pipelines; however, more advanced declarations are possible at runtime while determining precisely which pipeline to use.
static const TralfAnnotTag input_tags[] = {
    UOP_BB,
    TRALF_ANNOT_TAG_COUNT
};

static const TralfAnnotTag output_tags[] = {
    TRALF_ANNOT_TAG_BREAKPOINT,
    TRALF_ANNOT_TAG_COUNT
};

static TralfOpCheckResult *check_deps(...) {
    /* We depend on UOP_BBs covering the same interval
     * of trace as our breakpoint hits. */
    TralfOpCheckResult *cr =
        tralf_operator_check_result_new(num_deps);
    cr->can_run_now = true;

    for (size_t i = 0; i < num_deps; ++i) {
        cr->deps[i].filter.tag = UOP_BB;
        cr->deps[i].start_ts = deps[i]->start_ts;
        cr->deps[i].end_ts = deps[i]->end_ts;
        if (!i ||
            tralf_ts_compare(deps[i]->end_ts, cr->ts) > 0)
            { cr->ts = deps[i]->end_ts; }
    }

    return cr;
}

Figure 3.1: Dependency declarations for the breakpoint operator
While statically an operator only has the opportunity to declare which tags it produces as output, at runtime, an operator has more information available to it. First, the operator is given the identity of the trace file over which the query is running; this permits the operator to check whether preprocessing steps have been done that may be required by the operator. Second, the operator is given an accumulated set of dependencies that other operators (including the “top-level operator”, representing the user’s query) have requested; each such dependency represents a particular tag, a range of timestamps over which annotations with that tag are required, and optionally a filter which further restricts the set of annotations required in a tag-specific way (for example, a function call or return tag might be filterable on the identity of the function). Given these pieces of information, the operator examines the trace file and determines whether it is capable of satisfying the requested dependencies.

It is possible that some types of annotations might be able to be generated in a variety of different ways. For example, it might be possible to generate an annotation from a cache or precomputed index, or alternatively by direct computation on the underlying trace. Furthermore, in the case of a cache or precomputed index, it may be the case that the cache or index has been populated for a temporal subset of the trace, but not for the entire trace. In this situation, it is clearly desirable for the cache to be used over those time periods where it is available, falling back to direct computation where the cache is not yet populated. To implement such a system, two operators would be constructed, one of which reads from the cache and the other of which performs direct computation. The cache-reading operator would be given a higher priority, ensuring it would be selected when possible. To allow a single query to switch between operators based on intervals of time when the cache is and is not available, an additional mechanism is added to the dependency resolution system. If an operator is asked to satisfy some dependencies but detects that it can only satisfy those dependencies for a shorter period of time than requested, it may accept the request but declare the shorter time interval. When the time interval expires, the resolver must rebuild the pipeline, potentially discovering that the operator previously used is no longer able to satisfy the remaining dependencies and thus choosing an alternative operator. In addition, if an operator indicates that it is not able to satisfy the requested dependencies, it may optionally
declare a future point in time at which it believes it may be able to begin satisfying the dependencies; the resolver will build a fallback pipeline and run it to that point in time, at which point the pipeline will be rebuilt, hopefully including the now-applicable better operator.

### 3.3 Pipeline Construction

The automatic dependency resolver, when run, is given a *partial pipeline*, that is, a set of operators that are required to be present in the final pipeline. This set of required operators represents the query the application is attempting to perform, and these operators will typically not produce any output annotations but will declare all their input dependencies. The pipeline is constructed by means of the following algorithm.

First, the input and output tag lists for each operator (both those in the partial pipeline and all those registered as available for dependency resolution) are gathered and converted into equivalence classes. An *equivalence class* is a collection of annotation tags that always occur together in the output of an operator. All operators must fall into exactly one equivalence class; that is, all operators that share an output tag must all produce exactly the same set of output tags. This ensures that operators that share outputs are interchangeable with respect to their ability to produce output (they need not have the same dependencies nor be applicable to the same traces or time intervals).

Next, a dependency graph is constructed, in adjacency matrix form, between equivalence classes. Each equivalence class is represented by a node in the graph. An edge is constructed from equivalence class $C_1$ to equivalence class $C_2$ if *any* (but not necessarily *all*) operators outputting $C_2$ declare $C_1$ as a dependency. Cycle detection is then performed on the dependency graph; a cycle indicates that an equivalence class indirectly depends on itself, a situation which should be impossible. A dependency graph containing cycles is rejected and requires the operators involved in the cycle to be repaired before they can be used.

Third, a topological sort is performed on the equivalence classes using the dependency graph. This sort orders an equivalence class before all the equivalence classes on which it depends; that is, it ensures class $C$ appears before all classes...
consumed by operators that produce $C$. Additionally, for convenience, any equivalence classes produced by operators provided in the partial pipeline will be ordered before other equivalence classes, but only when this does not violate the topological ordering.

Fourth, the resolver instantiates all the operators in the partial pipeline and gathers their precise dynamic dependency information. During this process, if the partial pipeline contains more than one operator, any dependencies that can be resolved by other operators in the partial pipeline are removed from the collected set. For operators in the partial pipeline, because the application has clearly expressed its desire that the exact operators it specifies be used in the final pipeline, the possibility of an operator dynamically failing to satisfy dependencies it statically declares makes little sense; the only way to work around this problem would be to exclude the operator and include an alternative, a clear violation of the application’s stated intent. To avoid this possibility, the existence of an operator in the partial pipeline that fails to satisfy dependencies it statically declares is considered an error. Additionally, the resolver verifies at this time that two operators requested by the application do not output the same equivalence class; this would yield pointless and potentially-confusing duplicate annotations and most likely indicates an error in application logic.

Finally, once all operators provided by the partial pipeline have been instantiated and resolved, the resolver attempts to iterate the process of resolving the rest of the pipeline and executing it. These two actions may occur multiple times due to operators gaining and losing applicability and parts of the pipeline consequently needing to be rebuilt. To build the rest of the pipeline, the resolver simply iterates the equivalence classes while carrying a notion of “pending dependencies”, those that are not yet satisfied by the currently-constructed pipeline. For each equivalence class, if any pending dependencies belong to the class, the resolver first checks whether this equivalence class is provided by an operator already in the pipeline; this should be impossible and would indicate either an error in the topological sort or an operator dynamically declaring a dependency which it does not include in its static dependency set. The resolver then examines each operator whose output is equal to the current equivalence class, in order of decreasing priority. For each operator, the resolver queries the operator to check whether it is applicable to the set
of pending dependencies in the current equivalence class. If the operator accepts
the request, any new dependencies declared by the operator are added to the set
of pending dependencies and the resolver moves to the next equivalence class. If
the operator refuses the request, the resolver tries the next operator in the current
equivalence class. If all operators for a particular equivalence class refuse to satisfy
the pending dependencies, pipeline construction is impossible and the query fails.

Once no pending dependencies remain, the pipeline is fully constructed. The
resolver then executes the pipeline by injecting an annotation of type `BOOTSTRAP`.
This causes the last operator’s annotation handler to run (e.g., the trace file loader),
which examines the dependencies provided to it (e.g., a range of timestamps) and
generates the actual annotations (e.g., basic blocks). Each operator pushes its an-
notations further along the pipeline, causing the rest of the operators to be invoked
to examine the passing annotations and do their own processing. The dependency
resolver carefully crafts the dependencies requested of each operator such that the
pipeline will only run up to a desired target timestamp; for example, in the case of
a more-preferred operator being inapplicable but declaring a future time at which
it may become applicable, the resolver will only run the pipeline up to that fu-
ture time. Once such a future time is reached, the last operator will reach the end
of its dependencies and will stop generating further annotations; the pipeline will
naturally flush itself and give control back to the resolver, which may choose to re-
move some operators from the pipeline to swap in other operators before resuming
execution.
Chapter 4

Parallel Query Evaluation

Query execution is slow and traces are large. A trace of compiling a minimal Linux kernel runs to 106 gibibytes when recording kernel-mode execution only. Tracing userspace execution or tracing a long-lived server process could easily produce traces of many tebibytes. Running even the most computationally-simple query over such a large trace will necessarily take a lot of time. However, many of the queries Tralfamadore executes are easily parallelizable. For example, searching for all points in time where a particular line of code was executed can be trivially parallelized: one simply breaks the trace into a collection of disjoint time intervals and scans each interval.

I designed an architecture which makes parallelizing query execution straightforward. A query begins when a client application calls a function in libtralf requesting the query. The library makes a network connection to a demon speaking the tralfd protocol and sends the request. In the simplest case, and the only currently-working case, the network demon is simply an instance of Tralfamadore

![Figure 4.1: Parallel query execution architecture](image)
waiting for requests against a particular trace. However, it is very easy to insert a small transparent proxy\footnote{Such a proxy has yet to be written, however.} into the connection which is made aware of all the available Tralfamadore workers, themselves presented as demons. In this case, libtralf would connect to the proxy and send the request, which the proxy would then analyze. If the proxy understood the semantics of the particular request and knew how to parallelize it, it could rewrite the original query into a set of smaller queries and distribute them among the workers. If the proxy received a query it did not understand, it could simply select a lightly-loaded worker and forward the query to it verbatim.

4.1 Examples of Parallelizable Work

I herein present examples of how three typical tasks might be parallelized: breakpoint scanning, execution context scanning and ID generation, and backtrace generation.

4.1.1 Breakpoint Scanning

A breakpoint query is conceptually similar to a breakpoint in a traditional software debugger. The purpose of a breakpoint in a traditional debugger is to cause execution to stop when an instruction at a particular location in code is executed. Tralfamadore does post-hoc analysis and therefore cannot stop execution per se; a breakpoint’s purpose in Tralfamadore is thus to identify when the instruction was executed. Because an instruction may be executed more than once during a program’s lifetime, a breakpoint may be hit more than once. In a traditional debugger, these hits occur sequentially; after hitting a breakpoint, the user may order the debugger to resume the program under test, which will be stopped again the next time the breakpoint is hit. In Tralfamadore, breakpoint hits occur simultaneously; because all of execution is available, the result of a breakpoint query is a list of times at which the breakpoint was hit.

Without additional indices\footnote{It may be possible to speed up breakpoint queries by indexing function calls and returns and using debug information to determine to which function an instruction belongs; this does not eliminate the need for parallelism, only the applicability of this particular example.}, the only way to discover breakpoint hits is to scan
the trace from end to end. However, because each breakpoint hit is completely independent of any other breakpoint hits, this task is very easily parallelizable. The scheduling proxy can simply break the requested time interval into a collection of smaller time intervals and distribute them to workers. Each worker will return the set of breakpoint hits covered by its particular time interval. As long as the time intervals selected by the scheduler are disjoint and together cover the original interval, simply interleaving the individual workers’ responses as they arrive will yield an identical set of breakpoint hits to a sequential execution of the query, albeit in a permuted order.

4.1.2 Execution Context Scanning and ID Generation

The precise meaning of an execution context is given in Section 3.1. A trace will typically contain many contexts, with the system switching between contexts as interrupts arrive or processes are put to sleep by the kernel. Tracking these contexts is important for extracting semantic meaning from the system under test; although one may occasionally be interested in the interleaving of contexts, more commonly they should be treated as separate and independent entities each of which is analyzed on its own. To assist with handling context switches, we give each execution context that appears in the trace a unique context ID. This context ID is never shared with any other context, even if, for example, a thread dies and a new thread is given the same memory for its stack.

Of interest when dealing with context IDs is the ability to find the points in time at which context switches occurred, as well as discover which context ID was switched to at each such point. As context switches might be queried quite frequently, one could reasonably want to place them in an index so they can be retrieved quickly. However, to generate that index, it is necessary to scan the entire trace. It would be useful to parallelize this scan in order to reduce the delay between when a trace is taken and when analysis can start.

This problem is slightly harder than the breakpoint example in Section 4.1.1, because while the trace can be broken into intervals of time, the intervals are not independent. When a parallelized scanner observes a switch to a context it does not recognize, it cannot simply assume the context is new. Instead, the context might
actually be the same as a context that appears before the scanner’s assigned time interval; that is, the context might start in another scanner’s interval. Therefore, fix-ups are needed to ensure continuity of context IDs between time intervals assigned to different workers. A reasonable way to achieve this is to have each worker output intermediate data about the events it observes, essentially filtering the trace down to a much smaller size, then having a single postprocessing job convert the intermediate data into actual context IDs sequentially. Such a postprocessing job would be fast because it would not need to scan the entire trace.

4.1.3 Backtrace Generation

Generating backtraces is a very common operation when debugging software. In a traditional debugger, one backtrace is generated at a time, against the current moment in time. In Tralfamadore, because a breakpoint request generates every hit in the trace simultaneously, a backtrace request actually produces a collection of backtraces rather than just one. Further, the backtrace generated by a traditional debugger comes from walking the program under test’s stack, examining the chain of return addresses to discover the chain of functions that led to the current location. This can lead to incorrect output if the program under test has corrupted its stack. In Tralfamadore, because trace is available, a backtrace can instead be generated by ignoring the stack altogether and simply searching backwards in time for the execution of function calls. This mechanism cannot be confused by the program under test corrupting data. Backtraces are generated by using a function-invocation index to discover when functions are called and return, and looking for a sequence of function calls without matching returns leading up to the target point.

Because many backtraces may be requested simultaneously, parallelism is desirable. Backtrace generation can be parallelized very simply: by sending each breakpoint hit to a different worker. This is a very simple mode of parallelism, but it serves as a demonstrative example that breaking the trace into time intervals is not the only way to achieve query parallelism.
Chapter 5

Conclusion

Tralfamadore, as originally designed, provides a powerful framework for performing operating-system-level debugging and analysis. My work provides valuable additions to the system, enhancing its usability and performance.

First, the memory reconstruction tool provides the ability to, with very good performance, reconstruct the value of a variable at any given point in execution. This is one of the most common tasks ever performed while debugging a system, whether explicitly by ordering the display of the value of a variable or expression or implicitly by means of a watch or a higher-level debugger command. The address translation tool associated with the memory reconstruction tool provides the support necessary to efficiently deal with modern CPUs’ memory protection and translation hardware.

Second, the automatic dependency resolution system provides an important usability enhancement for tool writers attempting to use the Tralfamadore framework and its library of operators. Tool writers no longer need to manually determine the set of operators to assemble into an analysis pipeline, instead simply declaring dependencies and expecting these dependencies to be satisfied automatically. This allows a new tool writer to simply browse the set of available annotations and pick and choose which ones they wish to use, without having to trace backward through the other operators that need to be installed.

Finally, the parallel query evaluation framework provides a much-needed performance enhancement. While Tralfamadore’s architecture makes very powerful
analysis possible, and while deterministic replay reduces disk storage usage sub-
stantially, trace scanning will always be expensive. A modern CPU can run billions
of instructions every second; it is just not possible for a single computer to perform
complex analysis tasks on such a large quantity of data in a reasonable amount
of time. Only a cluster can provide this level of computational power; refactoring
Tralfamadore to work on a cluster is hence critical to making it useful for real-world
tasks. My parallel architecture provides an easy-to-deploy, piecewise approach to
parallelism in which any particular query can be parallelized if and when needed
without having to invest in parallelizing everything.
Chapter 6

Future Work

6.1 Memory Index

While the memory index has been implemented and used, it is not currently available in the most recent version of Tralfamadore, version 3, where the automatic dependency resolver is implemented. The memory index should be ported to Tralfamadore version 3.

The size of an epoch (50,000 writes) and the format of the index (a binary tree of nodes carrying simple run-length-encoded bitmaps) were chosen with the expectation that they would perform well. However, no rigorous verification of these choices has been made. It would be useful to evaluate the impact of changing epoch sizes in terms of the tradeoff between amount of trace scanned at a leaf node versus size of index file. It might also be useful to evaluate some alternative data structures. Finally, it would be informative to examine how the relationship between size and performance of memory index versus size of original recorded trace changes when the materialized trace file is replaced by a deterministic replay log, as this will both substantially reduce the size of recorded data (thus increasing the percentage overhead represented by the memory index) and also affect the speed at which an epoch can be replayed.

The current on-disk format is probably not well-suited to debugging systems with large amounts of physical memory, due to the run-length-encoded-bitmap format. It might be appropriate to consider alternative data structures that could re-
duce the size of the index; one possible option would be to replace the bitmap with a probabilistic hashtable. This would lead to a small percentage of false positives causing the reconstructor to do extra work, and might make the data less amenable to compression due to losing the benefit of the spatial locality exhibited by typical systems, but might be advantageous in some cases. It might even be useful to use RLE-compressed bitmaps on the lower levels of the tree, where memory writes are very sparse compared to the total size of memory and exhibit high spatial locality, then switch to probabilistic hashtables closer to the root where the larger time intervals represented by each node remove the spatial-locality advantage and where more total memory is written to.

6.2 Automatic Dependency Resolution

The dependency resolver has been implemented and tested, but has not yet been extensively used. More operators need to be ported to Tralfamadore version 3 before the dependency resolver will be particularly useful. Also, the ability of the resolver to use different operators to produce the same output types over different ranges of time has not yet been exploited. It is not yet clear how easy or hard it is to develop sets of operators that work together using this mechanism of transparently switching between reading caches or reading trace. Finally, the chosen mechanism for declaring dependencies has not been fully exercised; it may be that the dependency declaration mechanism is difficult to use when developing operators.

6.3 Parallel Query Evaluation

The parallel architecture has been designed but not fully implemented. The client library libtralf and the network daemon part of the backend engine are implemented, but the scheduling proxy described in Chapter 4 has not yet been written. Once the basic scheduling proxy is written, a vast number of future directions become available. The most obvious area of future work here is, of course, designing parallelization schemes for operators; each operator needs to be considered to determine how to parallelize it. Many will probably fall into similar classes, such as those that can simply have their time intervals subdivided and their output interleaved, but some will require more complex algorithms. Other areas of future work
here are higher-level concerns, such as how to efficiently utilize available local disk space on worker nodes when large numbers of traces are being worked on, and how to distribute jobs between workers given the distribution of trace files across workers. For example, some queries may be better served by going to a worker which has the trace on its local disk even if that worker is already busy, while other queries may be better served by going to a worker with ample processing time available even if it must access the trace over a network.
Bibliography


