RDB: A System for Incremental Replay Debugging

Michael W. Shapiro

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-97-12
July 1997

RDB: A System for Incremental Replay Debugging *

Michael W. Shapiro
Dept. of Computer Science
Brown University
mws@cs.brown.edu

May 19, 1997

Abstract

To facilitate the process of tracking program bug symptoms to their causes while debugging, we have developed a robust architecture for program tracing and replay. RDB is a framework for program instrumentation which modifies user programs to perform complete and highly efficient tracing of an execution including interactions with the operating system and asynchronous interrupts. We have developed a library which can be plugged into existing debugging tools to add trace-and-replay functionality, and we have developed a new prototype debugger using our library which provides replay control and flowback capabilities. We discuss the design and implementation of all aspects of the system and suggest future debugging tools and techniques based on this technology. This paper is the discussion of the author's project for the degree of Master of Science, and is the culmination of two years of work in this area with Professor Robert H.B. Netzer.[1]

1 Introduction

Debugging is the process of both removing defects from software and of understanding how a particular piece of software works. The debugger itself is a tool which aids the programmer in this process. Debuggers have historically been implemented as run-time or post-mortem analysis tools which provide static snapshots of a program's state at a given point during its execution, or of its state when some fatal error occurred. Some debugging tools also include the ability to perform run-time analysis of a program in order to detect particular types of program errors, such as memory access violations, memory leaks, or array boundary violations. These memory checking facilities are examples of debugger functionality which make some attempt at connecting a program behavior with its cause. The other types of standard debugging facilities, such as breakpoints, stack backtraces, and data formatting and display, are still only tools which the programmer uses while re-executing programs again and again in order to mentally re-connect complex sequences of events and logically traverse backward from a bug symptom to its cause.

*Partially supported by a grant from Siemens Corporate Research.
[1]Partially supported by ARPA order C-6-2753/D961, AFOSR Agreement F30602-96-2-0228.

We believe that the future of debugging tools lies in trace-based debugging. In our debugging model, the programmer can select one or more types of specialized program instrumentation, each performing some type of fine-grained monitoring during a program run. We believe that an essential building block required for advanced debugging tools is instrumentation for tracing the complete execution of a program so that it can replayed exactly in the debugger. Programs may be long-running and have arbitrarily complex interactions with their environment, and thus reproducing bugs by re-executing a given program may be impractical or impossible. We employ new fine-grained monitoring techniques to perform efficient program tracing. Our debugging framework can then use the traced information to replay this exact execution in incremental stages in the debugger. We have developed a new debugger, RDB, which can control program tracing and replay, and highlight connections between symptoms and their causes to aid the programmer in navigating through an execution trace.

In this paper we describe the design and implementation of the RDB debugging architecture, including our techniques for program instrumentation, tracing, and debugger control, and discuss in detail the implementation of the facility to trace a program's execution and then incrementally replay this execution in a debugger. We show how the architecture allows our instrumentation technology to be retrofitted onto existing debuggers while retaining all of their current features. Finally we discuss how our new prototype debugger exploits the ability to trace and replay a program's execution, and what types of interfaces to program replay we hope to develop in the future.

1.1 Related Work

The debugging framework we have implemented fits nicely within the debugging paradigm originally proposed by Agrawal [1]. Agrawal believed that there should be a systematic debugging paradigm based on execution *backtracking*. The SPYDER system he developed with DeMillo and Spafford [2] was a debugging tool which performed static analysis of a program in order to compute a *change set* of values which could potentially be modified by each statement so that the changes could be traced and restored. The PPD system also used the idea of compile-time analysis and program instrumentation and tracing to perform flowback analysis for debugging [3]. Both of these tools provided forms of execution replay, but were limited by the use of static analysis, which yielded unacceptable performance overheads and trace rates. Our work has built upon the same ideas and goals, while recognizing the need for tools with low overhead and trace rates, and the ability to record dynamic information including interactions with the operating system.

Previous work on dynamic analysis for program tracing has focused on the idea of intermediate process checkpoints. The IGOR system uses the operating system virtual memory facilities to periodically checkpoint the execution of a user program [5]. Past program states can be recreated from the checkpoints, but this technique traces orders of magnitude more data than necessary. Many techniques for improving incremental checkpoints have been proposed, including recent work by Plank, Xu, and Netzer on compressed differences [17]. Although such techniques can be used to improve the performance of dynamic tracing at the page granularity, Netzer and Weaver have shown that even lower overheads and trace rates can be obtained by dynamically detecting and tracing certain classes of memory accesses during program execution [16]. We finally achieve an efficient and complete implementation of these techniques with the work in this paper, and extend the instrumentation to support the type of systematic flowback analysis originally proposed by Agrawal.

2 Incremental Replay Tracing

The RDB architecture was initially developed with a specific problem in mind: we wanted to develop a working prototype of a trace-and-replay debugging system using our algorithms for incremental replay. We first motivate the trace-and-replay problem and review our work on it so far, and then discuss the implementation of an instrumentation library for RDB which implements one of our replay tracing algorithms.

2.1 Debugging Requires Replay

Debugging requires repeated executions of a program by the programmer in order to reproduce problems and obtain information which may help connect improper program behavior, or bug *symptoms*, with their causes. Traditional debugging tools require the programmer to repeatedly re-execute programs in the debugger, while providing little help at connecting problems back to their causes. With each re-execution, programmers typically waste time inserting more breakpoints, or adding more debugging output or assertions to the program itself. These techniques have several key disadvantages:

- User programs can be long-running, and may take extensive amounts of processing time to reach the point at which a bug symptom is exhibited; each re-execution in the debugger requires waiting for another lengthy execution to complete.
- User programs can have complex interactions with their environment, and thus it may be impossible to reproduce a bug during re-execution in the debugger. It may be difficult to re-create the exact pattern and timing of network connections which caused a server to crash.
- Post-mortem debugging can identify symptoms, such as a corrupted region of memory, but offers no way to determine *when* this region of memory was corrupted or by what routine, and thus provides only limited clues the programmer can use during a subsequent re-execution in the debugger.
- Writing additional debugging code wastes time re-editing and re-compiling programs, and the modifications may subsequently hide a race condition which caused a bug to manifest itself in the first place.

Programmers need tools which enable them to trace the execution of a program, including potentially complex system interactions, and then be able to exactly replay this execution in the debugger while using all of the existing debugging techniques and tools. The programmer should not only be able to replay an execution start-to-finish, but should be able to navigate through the execution forwards and backwards, restarting execution from any of a sequence of incremental restart points. Finally, the techniques employed to trace a program's execution must not be prohibitive in terms of run-time slowdown or in terms of trace volume. Our goal was to produce a prototype with a run-time slowdown of around a factor of two and trace rates which would allow runs of as much as a day in length to be stored on a gigabyte disk.

2.2 Unique-Spanning Reads

Previous work by Netzer and Weaver [16] identified a class of *unique-spanning* memory accesses which are a minimal set of memory accesses that must be traced in order to effect replay. First, a program's execution is viewed as a linear sequence of time intervals, each of some length T. A unique-spanning$_M$ read is the first read from a given address in an execution interval where this address was written in a previous interval but did not have its value traced in the M previous intervals. Their work demonstrated that dividing a program's execution into fixed-size time intervals and then adaptively tracing the unique-spanning reads within each interval can be done efficiently, and provides sufficient trace information for incremental replay. Experimental results [13] proved that this technique produced only 10–20 kilobytes of trace per second and met the goal of about a factor of two run-time slowdown. They further proved that tracing *unique-spanning*$_M$ reads was sufficient to provide *M-bounded* replay in the debugger. Replay is said to be M-bounded if we can replay any interval in the debugger by scanning at most the M previous intervals' trace data. Providing bounded-time replay of any interval and supporting short interval lengths, on the order of 10 or 20 seconds, are necessary for the debugging tool to be practical and useful. Previous work [19] has focused on developing state machines for efficiently detecting unique-spanning$_0$ reads. These machines are the most straightforward to design and implement because they produce trace data which allows us to replay an interval without having to restore the trace data for any previous interval.

1	*Read A*	Unique-spanning read
2	*Write A*	Write preceded by read
3	*Write B*	Write before any read
4	*Read A*	Read preceded by write
5	*Read B*	Read preceded by write

Figure 1: Example sequence of memory accesses in an interval

Since we are only going to use trace data for the current interval for replay, we can see that tracing unique-spanning$_0$ reads amounts to tracing the first read from a given address in each interval that is not preceded by a write to the same address. Consider the sequence of reads and writes shown in Fig. 1. For simplicity, assume each memory access is a full-word access and disregard system calls and interrupts for now. The rightmost column shows why only the initial value stored at address A needs to be traced for this sample interval. The sequence of accesses of the word at address A are *Read*, *Write*, *Read*. The first read is unique-spanning$_0$ because it is not preceded by any write, so we know that we must trace the value stored at A in order to replay. However, the second read of A does **not** need to be traced, because it reads the value which was written by the previous write to A. Assuming we properly restore the program context and address space and re-execute the same sequence of program code, write event 2 will write the same value it did when the program was originally executed. As a result, read event 4 will read the value from this write, and does not need to be traced. Using this same logic, we can see that since B is written first in the interval, read event 5 does not need to be traced. These observations are the key to producing low trace volume. We next need to build a fast monitoring mechanism to detect which reads need to be traced.

2.3 Monitoring Machines

In order to facilitate efficient run-time monitoring for unique-spanning reads, Netzer [13] proposed the idea of associating a finite-state machine with a small unit of memory, such as a word. Once the granularity of monitoring is chosen, the instrumentation code is designed to compute the address of a monitoring machine from the address referenced by each memory access, and perform an appropriate transition on this state machine. Previous work with tracing at the granularity of virtual memory pages produced impractical amounts of trace data; orders of magnitude more data than necessary was traced [5]. The monitoring machines use a comparatively large amount of storage during program execution, but pay dividends by yielding extremely fast instrumentation code and smaller trace sizes. We now examine the design of monitoring machines for replay tracing, and then consider how storage for such machines can be allocated and how efficient instrumentation code can be implemented.

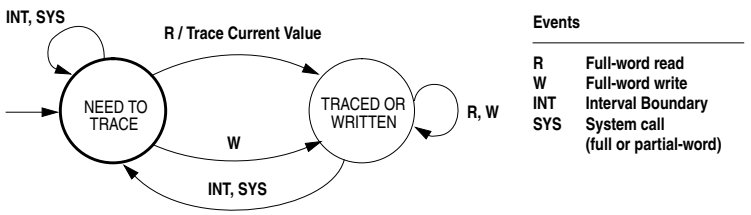

Figure 2: Two-state machine for tracing unique-spanning$_0$ reads

A two-state machine for detecting unique-spanning$_0$ reads which also handles system calls is shown in Fig. 2. The start state is denoted with a bold circle. This machine was the starting point for our work in designing a fast machine for replay tracing [19]. Before examining the evolution of the more complex machines, we revisit our simple example interval in Fig. 3, this time showing the state of the monitoring machines for the words at addresses A and B after each event. The transition which causes the value stored at address A to be traced is marked with an asterisk in the figure.

Event	Action	Machine for A	Machine for B
0	*Prior to interval*	Need to Trace	Need to Trace
1	*Read A*	Traced or Written*	Need to Trace
2	*Write A*	Traced or Written	Need to Trace
3	*Write B*	Traced or Written	Traced or Written
4	*Read A*	Traced or Written	Traced or Written
5	*Read B*	Traced or Written	Traced or Written

Figure 3: Example sequence of memory accesses in an interval

This machine suffers from several undesirable properties. The most critical is that an expensive action, tracing the current value, must conditionally take place on a particular *Read* transition. This implies that the run-time instrumentation must perform a conditional test and then possibly output a trace record at each memory load instruction. Furthermore, previous work [16] has shown that reads are the most frequent events. Our goal then was to devise a machine which has fast hooks, no conditional tests, and postpones all tracing activity until the end of each interval. We can observe that it is not necessary to actually the value stored at address A when read event 1 occurs. It is only necessary to save this value and remember that it needs to be traced. Furthermore, the

value only needs to be saved if it is overwritten. If we could arrange for values which have been read by a unique-spanning$_0$ read to be *buffered* if they are overwritten, we could examine each machine at the end of the interval and then decide whether tracing needs to be performed. We then either trace the buffered value, or the current value if it has not been overwritten.

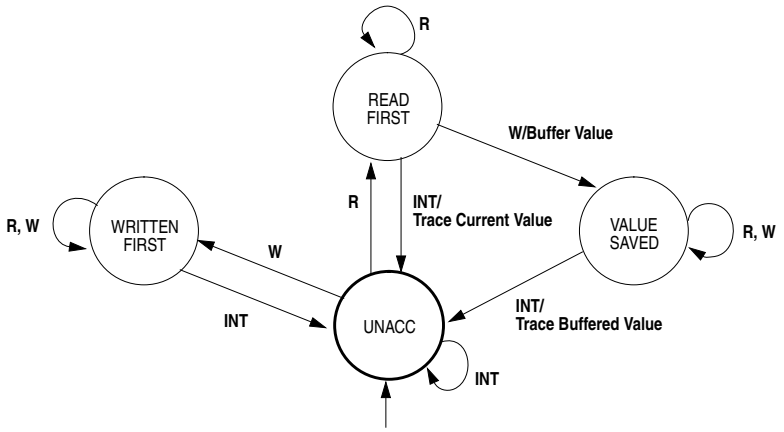

Figure 4: Four-state machine for tracing unique-spanning$_0$ reads

We developed the idea of using the copy-on-write virtual memory facility provided by the operating system to perform efficient, automatic buffering of values that we need to trace, but are overwritten prior to the end of the interval [19]. Prior to the execution of each interval, we execute the `fork(2)` system call to create a copy of the user process. The semantics of this call in Solaris and many other UNIX operating systems is that all of the process's virtual memory pages are not actually copied at the time of the `fork`. Instead, the pages are marked as copy-on-write, and the virtual memory system only copies when the parent or child faults on a given page [22]. Using this idea, we developed the monitoring machine shown in Fig. 4. By postponing all tracing until interval boundaries, this machine requires much less run-time overhead than the original two-state machine. Note that because of the automatic buffering of values by the virtual memory system, the read and write transitions perform **no** conditional tests. Also, we have added states to distinguish between values which must be traced and have been overwritten (the *Value Saved* state) and values whose current value can be traced (the *Read First* state). The start state has been renamed to *Unaccessed*; each word associated with a machine in this state has not yet been read or written in the interval.

2.4 System Calls

A system call can write a new set of values into the process address space which can not be reproduced during replay by simply re-executing the system call with the same arguments. Instead, we must arrange to trace the system call's return value and modifications to the address space so that we can restore these changes during replay, rather than actually re-executing the call. The use of a child process to automatically buffer overwritten values further complicates the handling of system calls. Since the child process is a copy of the parent, there is only one buffer slot for saving each word in the parent's address space. If a system call writes a word whose machine is already in the *Value Saved* state, and then the word is read following the system call, we need to trace *both* the previously buffered value and the current value written by the system call. This problem is exacerbated if the word written by the system call is then overwritten before the end

of the interval: the child process has the original value buffered, and now we also need to buffer the current value before it is overwritten. With this problem in mind, we considered two different approaches for tracing system call effects:

- We can simply trace all of the potentially affected addresses at the site of the system call. This means we have no problems with buffer space in the child, but we generate unnecessary trace data by immediately tracing all addresses affected by the system call.
- We can trace reads of values written by system calls. This reduces trace volume by only tracing those values written by the system call that are subsequently read. This approach has two complications: First, the system call hook must be able to identify if a value that already needs to be traced is going to be overwritten and either buffer or trace this value. Second, the system call traces will now be out of order in the trace file with respect to the order in which the system calls occurred, since subsequent reads may occur at any time. These out-of-order traces may complicate replay.

We considered monitoring machines for each of these approaches in [19], and selected a machine using the second method for our initial prototype. This monitoring machine is shown in Fig. 5. This machine includes transitions for full-word and partial-word loads and stores, system calls, and interval boundaries. We addressed the issue of buffering by concluding that performing a conditional test and potentially tracing at the site of the system call under some conditions was acceptable because system calls are very expensive compared to a single load or store instruction, and are much less frequent. We also concluded that by writing the trace data generated by reads following system calls to a separate trace file, we could sort this trace data prior to replay, thus solving the out-of-order problem. Therefore we must be able to distinguish values which should be traced to a 'standard' data trace file (the *Read First* or *Value Saved* states) and values which should be traced to the system call data trace file (the *Read After Syscall* and *Saved After Syscall* states. The transitions in Fig. 5 marked with an asterisk indicate traces following system calls.

2.5 Machine Implementation

We have developed techniques for writing efficient instrumentation code by implementing our state machines as small-depth *logical circuits* [15], avoiding the need to index into a transition table. By associating each machine state with a carefully selected binary value, each machine input can perform a transition by applying a simple function to map the integer representing the current state to the integer which represents the desired destination state. The bit layout we devised for the machine in Fig. 5 is shown in Fig. 6. We use five bits to represent the machine state, labeled A through E. This bit sequence is designed to translate into an efficient assembly language implementation. Bits marked with an asterisk indicate that the value of that bit may be either zero or one. Using this bit layout, the transitions for the various events can be implemented using the functions shown in Fig. 7.

This monitoring machine has several desirable properties: it implements system call tracing properly, it has an extremely fast instrumentation sequence for the very common read transition, and it has good overall run-time performance. But we still were not satisfied that we had done everything possible to optimize the write transition instrumentation. This code sequence, which we call a *hook*, was seven additional instructions per store in our SPARC implementation, much larger

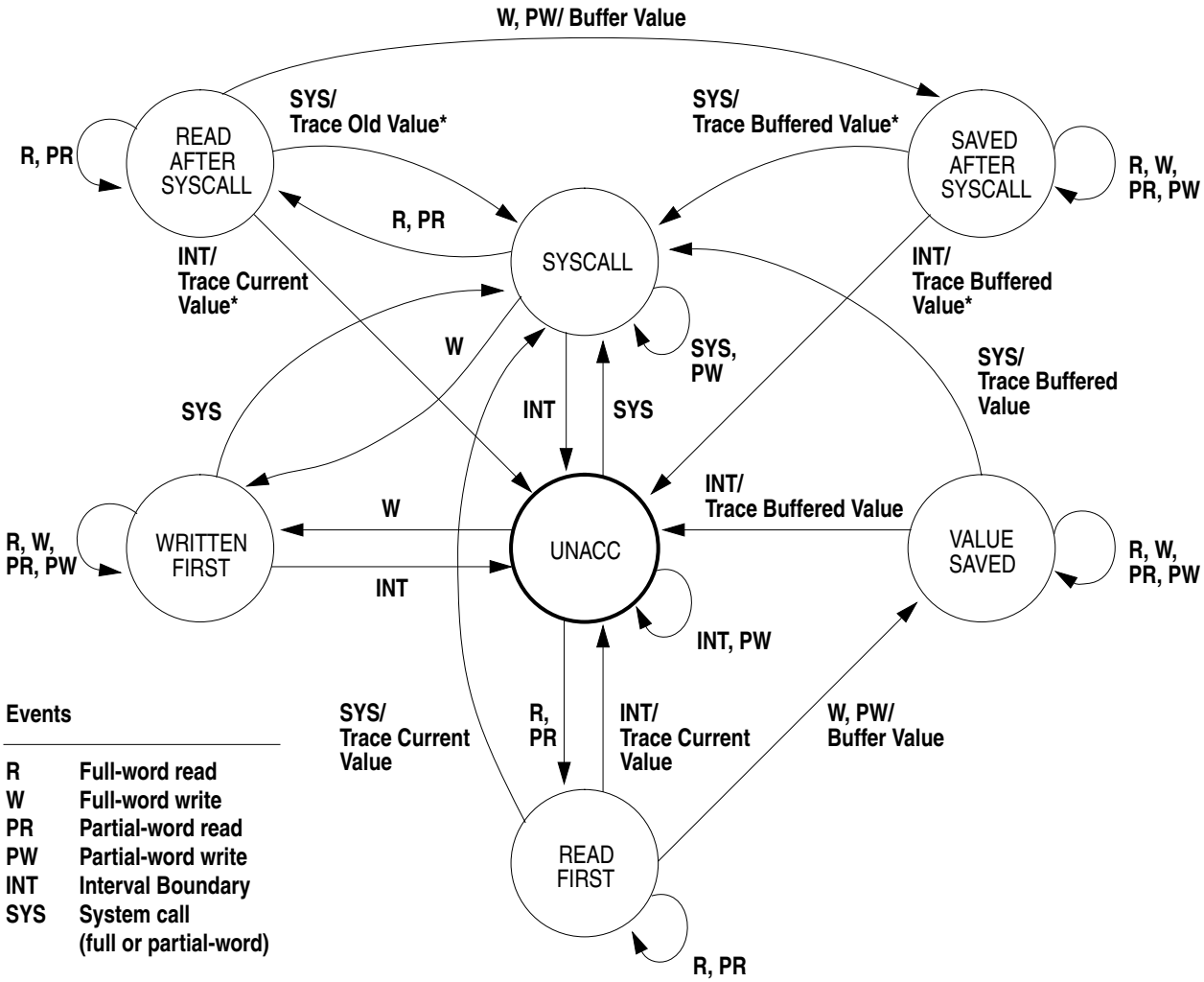

Figure 5: First complete monitoring machine for replay tracing

State	*Bit Sequence (ABCDE)*
UNACCESSED	01110
READ FIRST	00100
WRITTEN FIRST	100**
VALUE SAVED	000*0
SYSTEM CALL	011*1
READ AFTER SYSCALL	00101
SAVED AFTER SYSCALL	000*1

Figure 6: Bit assignments for monitoring machine states

Event	*Action*
READ	`(BD) = 00`
PARTIAL-READ	`(BD) = 00`
WRITE	`(A) |= (B), (BCD) = 001`
PARTIAL-WRITE	`(CD) = 01, (C) |= (B)`
SWAP	`(BC) = 00`
SYSTEM CALL	`(ABCDE) = 01101`
INTERVAL BOUNDARY	`(ABCDE) = 01110`

Figure 7: Actions to implement state transitions in monitoring machine

than the one or two instruction read hook. The length results from the complexity of the *Write* function: it must **or** two bits together, and then clear two bits and set another.

2.6 Improving the Machine

We were able to further optimize the machine by observing that the *Read First* and *Value Saved* states shown in Fig. 5 both denote a unique-spanning read. The reason for distinguishing these two states in our original design was that if a machine is still in the *Read First* state at the end of the interval, the user program can trace the current value at the address since it has not been modified. If the machine is in the *Value Saved* state, the value has been overwritten, and thus the user program has to fetch the buffered value from the child process and trace that instead. If we use the child process to trace all the values, then we can merge the two states, as shown in Fig. 8. The *Saved After Syscall* state mirrors the idea of the *Value Saved* state, and so this state can be merged into the *Read After Syscall* state.

If the child performs all of the tracing activity, this also reduces the need for sending values to be traced back and forth between parent and child, unless a system call overwrites a value we need to trace during the interval. In addition to reducing inter-process communication traffic, this machine has two desirable instrumentation properties. First, the partial-write transition is now always a transition back to the same state, thus *no instrumentation* is required for partial-word writes. Second, using the bit layout shown in Fig. 9, the write transition can be simplified so that it can be implemented in four additional instructions instead of seven as in our earlier prototype. We use three bits, which we call R for read, W for write, and S for system call. The logical actions associated with each event are shown in Fig. 10.

Further analysis of this condensed machine revealed that while we could now achieve a bit layout with faster write and partial-write hooks, we had lost a desirable property of the original machine: given a machine, it is no longer possible to tell whether or not the corresponding word has been modified in the current interval. The machine can now be in the *Read First* state if it has only been read first in the interval, or read first and then subsequently written. In the first case, the value has not been modified in the interval; in the second case the value has been modified in the interval. Although this property is not necessary for implementing the replay algorithm, we had previously shown that this feature can be used to implement *flowback debugging queries* [19].

The idea of flowback debugging is that while working in the debugger, the programmer may observe that a variable has an incorrect value. The programmer can then direct the replay engine to restore the state of the program to the point where this value was written, thus flowing backward

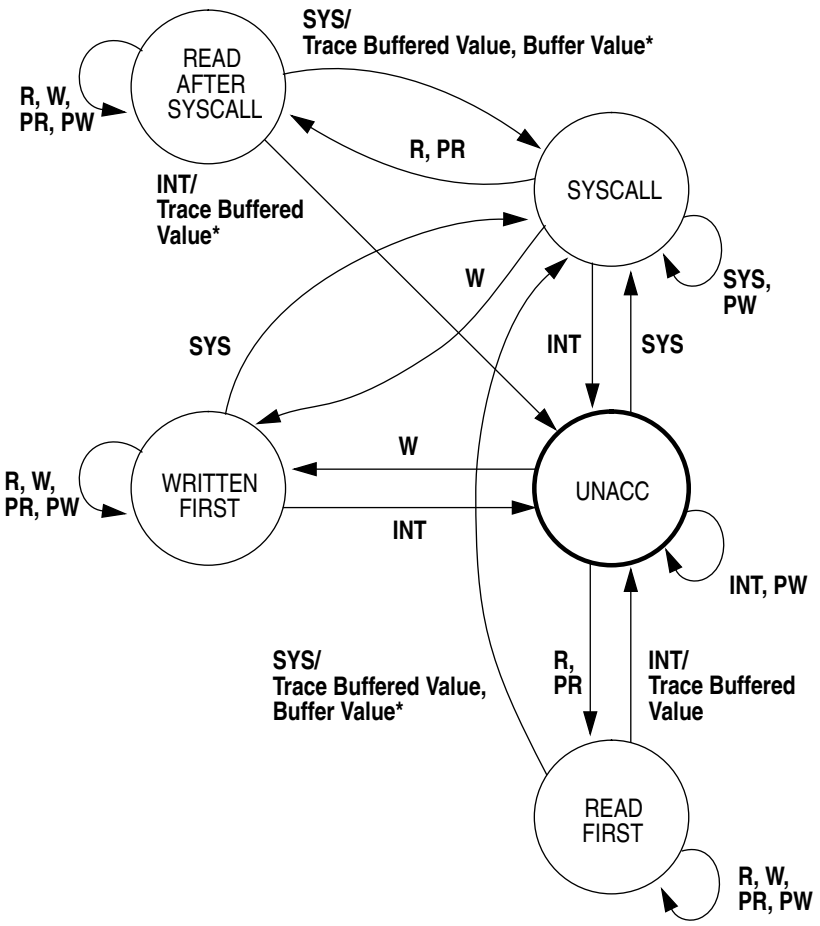

Figure 8: Simplified monitoring machine with *Value Saved* and *Saved After Syscall* removed

State	*Bit Sequence (RWS)*
UNACCESSED	100
READ FIRST	000
WRITTEN FIRST	*1*
SYSCALL	101
READ AFTER SYSCALL	001

Figure 9: Bit assignments for simplified monitoring machine states

Event	*Action*
READ	(R) = 0
PARTIAL-READ	(R) = 0
WRITE	(W) |= (R)
PARTIAL-WRITE	Nop
SWAP	(R) = 0
SYSTEM CALL	(RWS) = 101
INTERVAL BOUNDARY	(RWS) = 100

Figure 10: Actions to implement state transitions in simplified machine

from a bug symptom to a possible cause. We implemented this feature by limiting the bit layout for the machine to 16 bits, and then using the remaining 16 bits of the machine word to store the interval number in which the value associated with the machine was last written. At the end of the interval, we must be able to determine whether the word was written in this interval to decide whether we should write the completed interval number into the machine state word.

In order to restore this flowback property, we need to add states to distinguish unique-spanning reads whose values have or have not been subsequently overwritten. Our new machine and bit layout led to a new idea for how to do this and still retain the advantage of a fast write hook. The basic idea is to simply add another bit, which we label X, indicating whether or not the word has been modified in the interval. This allows us to add two additional states to the machine, *Partially Written* and *Read and Written*, but still keep the same write hook. The write hook is still fast despite the additional requirement of setting the X bit because of some assembly language tricks we describe later. This new machine is shown in Fig. 11, its bit layout is shown in Fig. 12, and the associated actions are shown in Fig. 13. This is the machine we used to implement our prototype replay debugger.

Once the state machine and state number assignments have been developed, we must consider two additional issues before the actual instrumentation code can be developed: the *granularity* of monitoring, and the *storage technique* used to record machine states. We briefly discuss our previous work on these two issues, and then describe the design of the trace-and-replay system and the associated instrumentation code.

2.7 Monitoring Granularity

Our work has focused on fine-grained monitoring, which for the replay problem implies monitoring of memory accesses at a finer granularity than virtual memory pages. Previous prototypes [15] have shown that performing replay tracing at the granularity of words approaches our goals for trace output and run-time slowdowns. Monitoring at a finer granularity, such as bytes, increases the amount of instrumentation code because common full-word accesses must perform transitions on multiple state machines. Monitoring at a coarser granularity causes unnecessary tracing, and also decreases the amount of information we can provide during debugging. The monitoring machine itself provides useful information to the programmer, since for each memory unit it indicates whether this address has been read or written since the previous interval boundary. If our monitoring is at a granularity coarser than that of words, we lose the ability to answer queries of the form "Has integer X been written yet?" since we cannot isolate state corresponding only to the word storing $X's$ value. As we will see, our state machines provide other information useful for controlling replay that we would like to associate with small units of memory such as words.

2.8 Run-Time Storage

Two methods of representing state machines have been considered: two-level bit vectors and flat bit vectors [13]. A two-level bit vector is similar to the idea of a virtual memory page table. The topmost bits of an address are used to index into a sparsely-allocated table, which then contains a pointer to a second-level table. Fig. 14 shows an example two-level bit vector implementation where a byte is used to store the state for each word. Fig. 15 shows an example flat bit vector implementation for the SPARC. In this approach, we literally reserve the entire bottom half of the

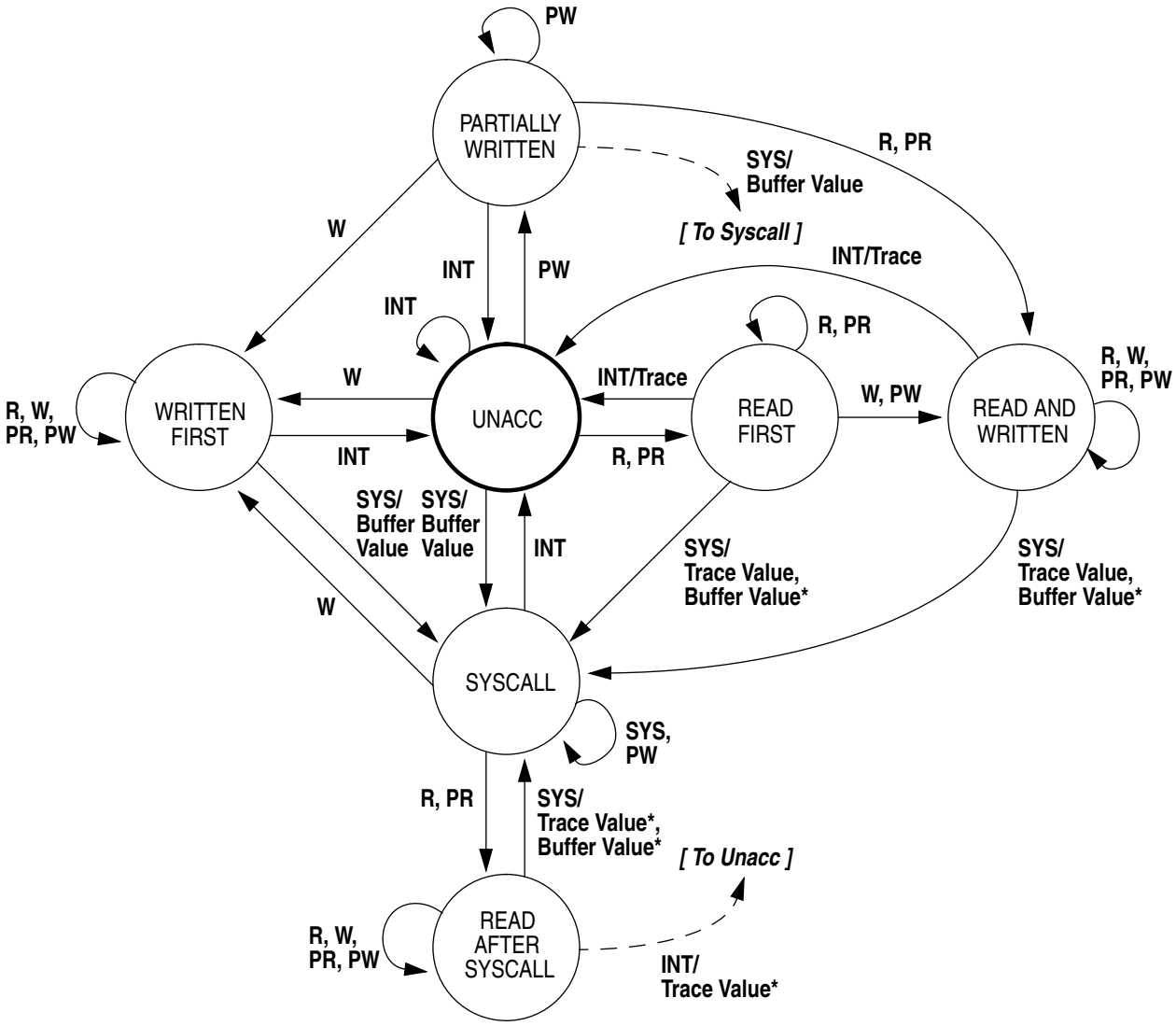

Figure 11: Improved replay monitoring machine with flowback property

State	*Bit Sequence (RWXS)*
UNACCESSED	1000
READ FIRST	0000
READ AND WRITTEN	0010
WRITTEN FIRST	*11*
PARTIALLY WRITTEN	1010
SYSCALL	1011
READ AFTER SYSCALL	0011

Figure 12: Bit assignments for final replay machine

Event	*Action*
READ	(R) = 0
PARTIAL-READ	(R) = 0
WRITE	(W) |= (R), (X) = 1
PARTIAL-WRITE	(X) = 1
SWAP	(R) = 0, (X) = 1
SYSTEM CALL	(RWXS) = 1011
INTERVAL BOUNDARY	(RWXS) = 1000

Figure 13: Actions to implement state transitions in final replay machine

address space for bit vector storage. A word is used to store each machine's state, so we can map from an address to its machine by simply subtracting off the highest bit.

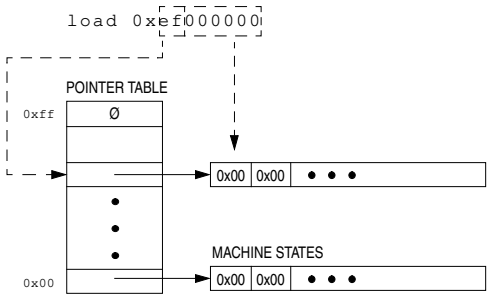

Figure 14: Two-level bit vector using one byte to encode the state for each word

The advantage of selecting the flat bit vector approach is that the mapping from a word to its machine storage is so simple to implement, we can fold it into a single instruction using an addressing mode which adds two registers. In cases where this is not possible, we only have to pay one additional instruction. The two-level approach requires more instrumentation code, which will increase our run-time slowdown. The disadvantage of the flat bit vector approach is that it appears much more limiting to the user program to offer it only half the address space. We do not believe that this is so limiting as to preclude its use in a real system. First, storage space in the form of virtual memory pages is allocated sparsely in the bottom half of the address space just as in the top half for the user program, so the program only consumes twice as much physical memory or swap space as it would have, and process memory requirements are typically small when compared with the available amount of swap and physical memory available on the machine. Second, since a 32-bit address space offers 4 gigabytes of addressable memory, we are only limiting the user program to 2 gigabytes, which typically far exceeds the needs of user programs except for those which map huge files directly into memory. Finally, 64-bit operating systems are just beginning to emerge as a standard, so this technique will offer even more addressable memory to programs on future architectures.

Although the flat bit vector requires us to reserve half of the user address space for machine storage, we only allocate virtual memory pages in this region as needed. Using the RDB facility to intercept signals, discussed in detail in the next section, we detect faults in the reserved region of the address space, which we call the *mirror region*. The instrumentation library's signal handler then maps a new page of anonymous memory in at the virtual memory page boundary nearest to the fault, and initializes each word of the page to the *Unaccessed* state. The handler then returns

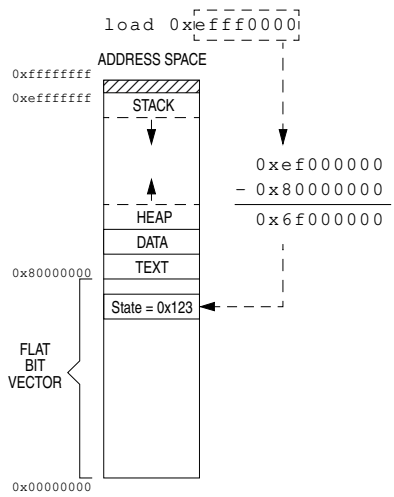

Figure 15: Flat bit vector effectively dividing the address space in half

control to the user program, where the instrumentation code now restarts and accesses a valid machine state. Thus we only double the actual size of the process in memory, not its potential size, and the pages of machine state storage exhibit the same locality of reference as the pages associated with the user program.

3 Program Instrumentation

We now turn to the implementation of our system for trace-and-replay debugging. We first discuss the general framework provided by RDB for developing instrumentation tools, and then examine the implementation of an RDB library to implement the trace-and-replay algorithm described so far. In order to trace a program's execution and be able to exactly replay this program in the debugger, we need to add instrumentation to a program to detect and trace several different classes of events, including:

- *Memory Accesses* Our techniques for execution tracing rely on the ability to perform fine-grain monitoring of events in a program's execution, such as all reads from and writes to memory. Thus we need to ability to add instrumentation code at the assembly language or object code level to all instructions of a given class, such as loads or stores.

- *System Calls* A program's interaction with the operating system and with other processes and devices occurs through the use of system calls. In addition to intercepting system call traps, we are also interested in detecting modifications to the user program's address space which occur as a result of the system call.

- *Synchronous Exceptions* Synchronous exceptions, such as memory access violations, invalid instructions, and floating point arithmetic exceptions, are delivered as signals in UNIX. In order to correctly replay a program's execution, we need to be able to intercept and record such events. UNIX processes may also install handlers for such exceptions, so we also want to make sure we trace whatever actions the process takes in response to the event.

- *Asynchronous Exceptions* UNIX processes may also deliver asynchronous exceptions to one another as signals. This mechanism is used to abort programs (such as when the user issues a `kill` command), and is used by the operating system to inform the user process of particular events or conditions. We need to intercept and detect such events, and we also need to identify exactly when the event occurred. Specifically, we must be able to record the unique execution instance of the instruction which was interrupted by the exception so that it can be re-executed in the debugger at the proper time.

The RDB architecture takes a two-phase approach to program instrumentation, as shown in Figs. 16 and 17. First, the assembly code following compilation and optimization is run through an *Assembly Language Translator* utility (ALTR) which follows a set of rules describing how to add instrumentation code to instructions of interest. Second, a run-time engine (`librdb_trace.so`) is linked with the executable in order to provide hooks to intercept events such as system calls and exceptions. An instrumentation-specific run-time library is also linked into the executable along with `librdb_trace.so` in order to provide the implementation of a particular instrumentation algorithm. The run-time engine provides a mechanism whereby the instrumentation-specific library can register callbacks to be invoked when events of interest occur during the program's execution. We will subsequently refer to these libraries as the *tracing framework* and the *instrumentation library*.

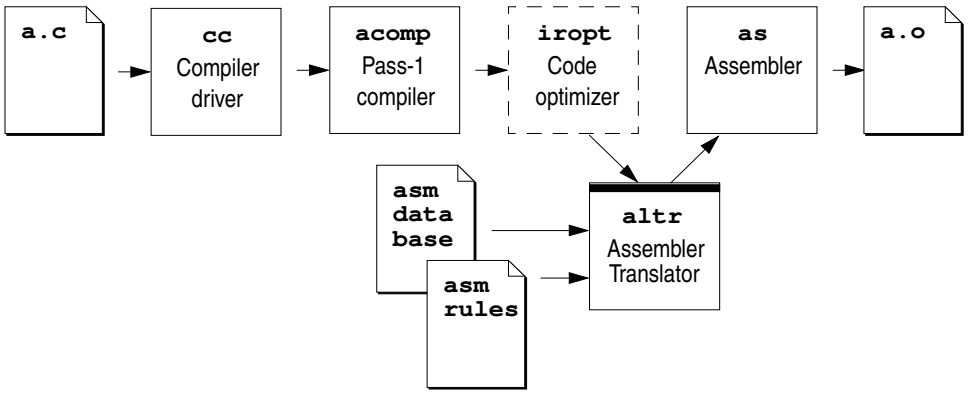

Figure 16: RDB program instrumentation phase for the Sun C compiler

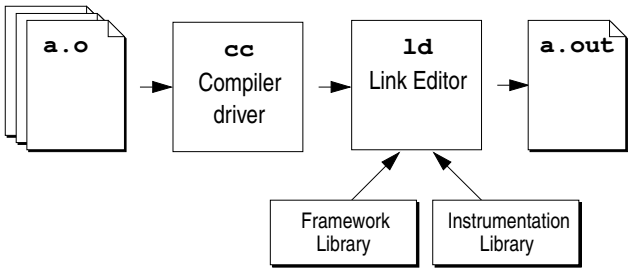

Figure 17: RDB program instrumentation linking phase

3.1 Assembly Language Translator

We developed our own assembly-to-assembly translator, ALTR, for the Sun Solaris SPARC assembler. The translator is capable of accepting assembly output from a variety of sources, including compilers for several different languages from Sun and the GNU project, and hand-written assembly code. We examined several available tools for instrumenting object code on the SPARC, but found that none was simple, flexible, and robust enough to suit our needs at the time. Our primary interest at the outset of development of the replay debugging system was the ability to experiment: as we discuss later, there were many iterations of our tracing algorithms, each with associated assembly code instrumentation. Also, our goal was to create an architecture where new types of instrumentation could be easily developed. To meet this need, we designed ALTR to accept a *rules file*, which consists of a small set of keywords to specify a type of instruction in the assembly input, and then inline assembly code to replace the matched instruction in the output.

Object code or assembly instrumentation can performed at one of three levels:

1. The assembly source code from the compiler can be instrumented before it is assembled into an actual object file, and then the normal linking phase produces an instrumented executable.
2. The executable file can be instrumented following assembly and linking, producing an instrumented executable.
3. The executable's text segment can be patched live by a debugger once it has been loaded into memory, just prior to the start of execution of the user process.

We discarded the ideas of instrumenting a linked executable or a live process for much the same reason: we wanted to simplify our implementation, and make it easy to extend and debug. In addition, since we wanted to locate instructions of interest using pattern matching or instruction names, and add instrumentation using native assembly language syntax, the development of object-based tools would have required the additional development of a disassembler to first convert object code to assembly code to perform pattern matches, and then code to re-assemble and integrate instrumented sequences on the fly. Instead, by adding instrumentation at the assembly source level, we work directly with the input assembly source and then just invoke the standard assembler afterward. With these design decisions made, we were able to implement the complete translator utility in just over 2,000 lines of Perl [23] code.

```
name = "sample-rule", \
instr = $ATTR_LOAD {
  add %g2, 1, %g2 !  Increment %g2
  ${0}
}
```

Figure 18: ALTR rule for incrementing a register prior to each load

An example translator rule is shown in Fig. 18. This rule increments a register we have reserved for instrumentation purposes, *%g2*, prior to each instruction which loads from memory. This example illustrates how ALTR allows powerful instrumentation rules to be written very concisely.

The `instr = $ATTR_LOAD` syntax specifies the class of instructions to match. The `$ATTR_LOAD` constant tells ALTR to match instructions whose opcodes have the memory load attribute defined in the instrumentor's internal database of all known SPARC opcodes. The `${0}` syntax is a convenience variable which indicates that the translator should substitute the text of the matched instruction itself. Other convenience variables allow the substitution of instruction arguments or matched regular expressions in an instruction or its argument list.

Besides parsing the rules file and performing lexical matching and substitution, ALTR also automates a number of complicated instrumentation tasks for the developer. One task is to handle matched instructions which are currently located in the delay slot of a *delayed control transfer instruction* [6]. If such a matched instruction is to be replaced with a sequence of more than one instruction, then ALTR automatically moves the entire sequence out of the delay slot, while ensuring that the new instruction sequence will only execute in the same code paths as the original instruction would have before the transformation.

In addition to assembly language transformations, ALTR is also aware of symbolic debugging information present in the assembly input file in the form of `.stabs` and `.stabn` pseudo-operations. These directives are output by the compiler along with the assembly code for each source file to record variable type and storage information, and to record the relationship between assembly instructions and source code lines. ALTR includes special processing code for the `N_SLINE` line number directives output by the Sun and GNU compilers. Each `.stabn N_SLINE` directive specifies a source line number, and then an offset computed from a label marking the first assembly instruction associated with this source line. In order to ensure that standard debugging tools can correctly single-step through an instrumented program, ALTR must be aware of matched instructions which are referenced by a `N_SLINE` directive. If additional instrumentation instructions are added prior to the labeled instruction, the `.stabn` directive is modified to refer to the first instruction in the instrumented group. As a result, the instrumentation code and original instructions are executed together during a single-step in the debugger.

Finally, we added to ALTR the capability to produce additional `.stabs` directives during its instrumentation pass to record the offset within the text segment of the start and end of each sequence of instrumented instructions. Following linking, our link-editor wrapper script invokes a tool which processes the instrumentation offset debugging information, and adds a new section to the output executable named `.iot`. This section contains an *Instrumentation Offset Table*, which is a sorted array of the program counter ranges that correspond to instrumented code at run-time. We provide a set of utility routines in the framework to allow the instrumentation library to load this table. When an asynchronous event occurs at run-time, the instrumentation library performs a binary search using the contents of the *%pc* register to determine if a sequence of instrumented code was interrupted by the event.

In order to provide accurate tracing and replay of a UNIX program, we must also provide instrumented versions of the necessary system libraries. This requirement forced us to prove the robustness of ALTR: using our licensed copy of the Solaris source code, we were able to compile our own instrumented version of the Solaris standard C library, including all C code and hand-written assembly code.

3.2 Run-Time Tracing Framework

We now discuss the design and implementation of the framework library which is linked with the executable after each source file has been compiled and instrumented by ALTR. The tracing framework, implemented by the shared library `librdb_trace.so`, provides a set of utility functions for the instrumentation developer. The library also provides its own versions of common library and system calls found in `libc`. This is necessary because in the final version of an instrumentation tool, even `libc` itself will be compiled using the instrumentation, and thus the instrumentation run-time library may need to call uninstrumented versions of certain system calls and library calls internally. In addition, the framework allows the instrumentation library to register callbacks to be invoked when certain events occur, including:

- *Process Initialization* The library can receive a callback prior to the start of the process's execution of `main`. As we will see shortly, this issue is slightly complicated on Solaris due to the presence of the run-time linker `ld.so.1` and `.init` sections. This callback allows the instrumentation library to initialize data structures, and register other callbacks and timers.
- *Process Termination* The library can receive a callback just prior to process termination, which we define as either just prior to the trap into the kernel to execute the `_exit` system call, or just prior to the delivery of a UNIX process signal whose delivery is known to terminate the process. This callback can be used to write out buffered trace data, tear down data structures, or compute statistics.
- *Process Signals* At any time, but typically in the process initialization callback, the instrumentation library may register a callback to be invoked when a UNIX process signal is delivered. The callback receives detailed information about what type of signal or exception occurred, and a pointer to the interrupted context. Following processing in the callback, the instrumentation library may choose to have the framework continue with the delivery of the signal to the user process, in which case all the usual user signal processing occurs, or to indicate that this signal was internal to the library and program execution should resume as if the signal never occurred.

3.2.1 Process Initialization

Process startup on Solaris begins not in code in the actual executable, but rather in code in the run-time linker `/usr/lib/ld.so.1` [8]. This program is responsible for mapping in all of the shared libraries upon which the user program depends, performing relocations if necessary, calling any initialization functions provided by the shared libraries, and then transferring control to the `_start` routine embedded in the executable [21]. This routine constructs the initial stack frame, invokes the `_init` function which calls library and static C++ object initialization functions, and then calls `main`. This routine also installs two exit handlers using the standard C library `atexit(3c)` routine: one to invoke the executable's `_fini` function to destruct static C++ objects and call other cleanup functions, and another to invoke a function inside of `ld.so.1` to do the same for each shared library.

With this boot process in mind, we carefully redesigned `_start` in order to provide the proper hooks for our library. We decided to take control of the process in `_start` because the framework itself is implemented as a shared library, and thus we need to wait until `ld.so.1` has done its job. Pseudo-code for our version of `_start` is shown in Fig. 19. The actual routine is implemented

directly in assembly language. This routine is pre-compiled and provided in the `crt1.o` object file which is linked into the executable by the compiler driver. Since we have our own compiler driver, we simply substitute our own `crt1.o` in place of the one specified by the Sun or GNU compiler driver.

---

void _START(void)

1 initialize first stack frame
2 locate *argc*, *argv*, *envp* on stack

/* *%g1 is set to the address of ld.so.1's cleanup function* */
3 _rdb_init(*envp*, *%g1*);

4 atexit(_fini);
5 _init();

6 *result* = main(*argc*, *argv*, *envp*);
7 exit(*result*);

Figure 19: _START algorithm

The RDB `_start` routine first invokes the startup function `_rdb_init`, which is provided by the framework shared library. This function is responsible for initializing the framework itself, and then invoking the instrumentation library initialization callback. This must occur prior to the calls to `atexit(3c)` and `_init` because these functions themselves may be instrumented, and will thus depend on the instrumentation library being initialized. We pass the environment pointer to the framework so that it can decode arguments and options we allow to be specified as environment variables, and we also specify the function pointer contained in SPARC register *%g1*, which if non-zero, is the address of `ld.so.1's` cleanup function. This must be passed to the framework and not installed via `atexit(3c)`, as in the standard `_start`, because during execution replay, we may stop the process following the execution of `exit(3c)` but prior to the `_exit(2)` trap, and then direct the process to flow backwards and replay an earlier point in the execution. If we allowed `ld.so.1's` function to be called during `exit(3c)`, then shared libraries could potentially be destroyed and unmapped, and then would have to be re-mapped in order to continue replay. Thus we avoid calling the cleanup handler until the final termination of the process.

3.2.2 Process Termination

UNIX processes can terminate in one of two ways: they can be terminated as the result of the delivery of an unhandled signal whose defined behavior is to terminate the process, or they can be terminated when the process calls the `_exit(2)` system call. In order to clean up properly and call the instrumentation library's termination callback, we arrange to intercept both of these events in the tracing framework. Signals are intercepted as part of the framework's signal architecture, described in more detail next. We intercept calls to the `_exit` routine provided by `libc` by *interposing* [21] our own version of this routine in front of the one exported by `libc`. The `_exit` provided by the framework library first calls the instrumentation library's termination handler, then cleans up data

structures internal to the framework, and finally uses its own uninstrumented version of `_exit` to trap into the kernel and terminate the process.

3.2.3 Process Signals

During initialization, the framework library uses an internal version of the `sigaction(2)` system call to install its own signal handler for all valid UNIX process signals. The framework also interposes its own version of `sigaction` and other signal management functions in front of those provided by `libc`. This allows the framework to intercept all signals as they are delivered to the process by the kernel on the one hand, and to intercept all modifications to signal handlers and dispositions by the user program on the other. When the user program installs a handler for a given signal, this makes no change to the array of signal handlers stored in the process `uarea`. Instead, our interposed `sigaction` function modifies a similar array stored inside the framework library.

The framework sets up its signal handlers so that all other signals are blocked on entry to the handler. The framework can then invoke the appropriate signal callback installed by the instrumentation library without interference from other signals. If the instrumentation library callback returns a status indicating the signal was handled internally, the framework restores the signal mask and interrupted context and the user process is "unaware" that the signal ever occurred. Alternately, the instrumentation library may allow the user process to receive the signal. From the perspective of the user process, the signal may have one of several defined dispositions:

- *Blocked* The user process may block delivery of the signal for later delivery. The framework keeps track of pending blocked signals and can deliver them later if the signal is unblocked, similar to how the operating system handles this condition.
- *Ignored* The user process may mark the signal as ignored; the event is discarded with no further processing.
- *Handled* The user process may install a signal handling function to be invoked when the signal is delivered. This handler may have a corresponding signal mask to be set during execution of the function, so that the handler itself may or may not be able to be interrupted by other signals.
- *Default* The user process may allow the operating system to proceed with the default behavior associated with the given signal. These default behaviors are typically: ignore the signal, stop the process, terminate the process, and terminate the process and write out a core dump.

This last behavior is the most difficult to emulate. Although we can terminate the process by simply calling `_exit(2)` from inside the framework's signal handler, this will not produce a core dump. In addition, we need to be concerned with two other issues: a UNIX process has an exit status encoding why the process terminated which can be obtained by a parent waiting for it. Also, if a core dump is written, we want the stack at the time of termination to reflect the location in the user program which initially was interrupted by the signal, as opposed to the stack trace reflecting our presence in the framework's signal handler. We must preserve these behaviors to allow the process to cooperate with other processes and to allow the user to use existing postmortem debugging tools effectively.

We solve this problem in the framework signal handler by keeping track of the defined default behavior of each signal. If we need to stop, terminate, or core dump the process, we first raise the original signal in the process's signal mask using the `kill(2)` system call. This signal will not be immediately delivered to the process since we arranged to enter the framework signal handler with all signals blocked. At this point the signal is pending but cannot be delivered. We then modify the interrupted context, which contains the set of blocked signals at the time the signal was originally delivered, to have all signals blocked except the one which we are currently handling. Finally, we remove the framework's signal handler for this signal, and restore the context. The program's context is restored to the original interrupted context, but before the kernel returns the program to user mode, it realizes that this signal is pending, has the default disposition, and all other signals are blocked, so the original signal is now re-delivered with the desired consequences.

4 Tracing Prototype Implementation

We implemented a prototype trace-and-replay instrumentation library for RDB running under Solaris for the SPARC architecture, using the flat bit-vector storage approach described earlier. We now discuss the implementation details of the run-time instrumentation code and the library code for tracing, and present performance results for our prototype. For this work we only provide support for single-threaded processes. Support for multi-threaded user processes involves additional instrumentation, including recording scheduling decisions, and we leave this as an open issue for future research. We also only trace a single process from start to finish. If this process forks another UNIX process, we continue tracing in the parent process and disable tracing in the child. We hope to implement improved support for forking child processes in the future.

4.1 Bit Layout

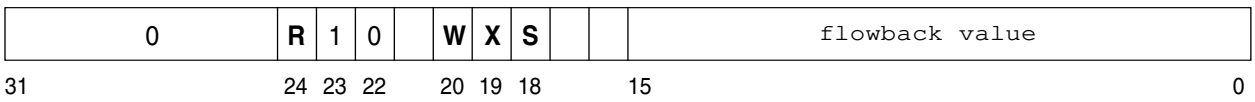

Figure 20: Bit layout for replay machine state

The bit layout we devised for representing the state of each machine is shown in Fig. 20. We use the lower 16 bits (half-word) of our machine state word for the *flowback value*. We will discuss the meaning of this value shortly. The upper half-word stores the bits which represent the machine state, labeled R, W, X, and S. We also define several other bits to be fixed at either one or zero. This layout is specifically designed to optimize first the read hook, and then the write hook. The SPARC allows memory to be accessed at the granularity of byte, half-word, full-word, and double-word. Any manipulation of data must take place in a register. Since our read hook needs only to zero the R bit, we isolate this bit in the high-order byte. This means that with a *single instruction* we can store a byte from register *%g0* (which always contains zero) to the machine address, thus implementing the read hook. For the write hook, we need to **or** the R bit with the W bit, and then set the X bit. The bit layout allows a very efficient implementation of these two operations, as shown in Fig. 21. By loading the upper half-word of the machine state word into the lower half of a register, and then shifting this over 4 bits and **or** ing the shifted bits with the original register, we perform both operations simultaneously while preserving both the R and S bits.

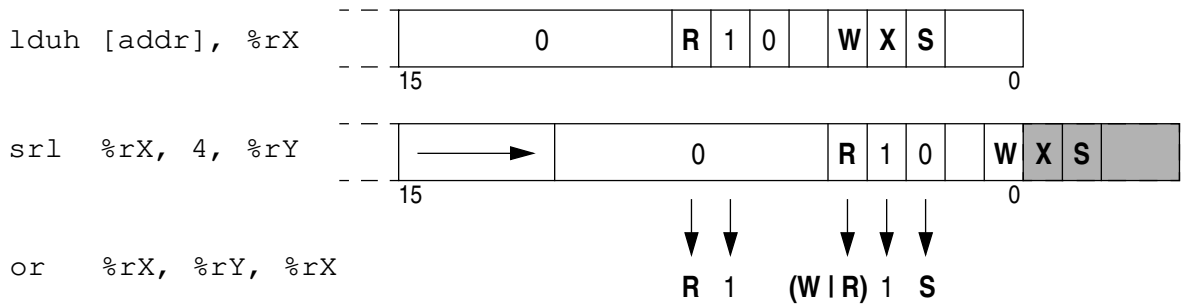

Figure 21: The bit layout lets us compute `W |= R` and `X = 1` simultaneously

4.2 Instruction Hooks

A complete set of sample instruction hooks for full and partial-word reads and writes, and the two SPARC atomic load-store instructions are shown in Fig. 23. This is not the complete set of hooks: slight variations on the presented hooks are also needed to handle loads and stores where the matched instruction uses an addressing mode which sums two registers or sums a register and a 13-bit immediate value. In each case, an additional instruction is required to pre-compute this sum in a temporary register. Throughout the instrumentation code, we rely on using several of the SPARC global registers, labeled *%g0* through *%g7*, as temporary storage and to store the constant 0x80000000. The SPARC application binary interface reserves registers *%g2*, *%g3*, and *%g4* for the user program, and we use our instrumentation front-end to invoke the underlying compiler with a flag telling it not to generate code for these registers so we can use them. Registers *%g5*, *%g6*, and *%g7* are reserved for the operating system implementation, but our analysis of Solaris on the SPARC reveals that *%g5* and *%g6* are already used by other user-level debugging tools, such as the Sun *Dbx* memory access checking instrumentation, and that *%g7* is used to point to thread-specific data in multi-threaded programs. Thus since we are a debugging tool, and we do not support trace-and-replay for multi-threaded user programs at this time, we are free to use registers *%g2* through *%g7* in our instrumentation. The assignments we chose are shown in Fig. 22.

Register	*Use*
%g0	Zero
%g1	Volatile (used in PLT)
%g2	Temporary for instrumentation
%g3	Temporary for instrumentation
%g4	Constant value 0x80000000
%g5	Temporary for instrumentation
%g6	Constant value 0x80001fff
%g7	Software Instruction Counter

Figure 22: Global register assignments for SPARC prototype

4.3 The Stack

As discussed in our earlier work [19], the SPARC also has special `save` and `restore` instructions to manage *register windows*, or program views of the actual registers provided by the hardware. Depending on the number of active register windows at the time of a `save` or `restore`, a *window*

Original Instruction	*Instrumented Code*	*Comments*
`ld [%rY], %rX`	`stb %g0, [%g4+%rY]`	`! (R) = 0`
	`ld [%rY], %rX`	`! Full-word read`
`ldub [%rY], %rX`	`andn %rY, 0x3, %g3`	`! Word-align address`
	`stb %g0, [%g4+%g3]`	`! (R) = 0`
	`ldub [%rY], %rX`	`! Partial-word read`
`st %rX, [%rY]`	`st %rX, [%rY]`	`! Full-word write`
	`lduh [%g4+%rY], %g5`	`! Load state bits`
	`srl %g5, 4, %g2`	`! Shift right 4 bits`
	`or %g5, %g2, %g5`	`! (W) |= (R), (X) = 1`
	`sth %g5, [%g4+%rY]`	`! Store state bits`
`stb %rX, [%rY]`	`andn %rY, 0x3, %g3`	`! Word-align address`
	`stb %rX, [%rY]`	`! Partial-word write`
	`lduh [%g4+%g3], %g2`	`! Load state bits`
	`or %g2, 0x8, %g2`	`! (X) = 1`
	`sth %g2, [%g4+%g3]`	`! Store state bits`
`swap [%rY], %rX`	`lduh [%g4+%rY], %g2`	`! Load state bits`
	`or %g2, 0x8, %g2`	`! (X) = 1`
	`andn %g2, 0x100, %g2`	`! (R) = 0`
	`sth %g2, [%g4+%rY]`	`! Store state bits`
	`swap [%rY], %rX`	`! Full-word swap`
`ldstub [%rY], %rX`	`andn %rY, 0x3, %g3`	`! Word-align address`
	`lduh [%g4+%g3], %g2`	`! Load state bits`
	`or %g2, 0x8, %g2`	`! (X) = 1`
	`andn %g2, 0x100, %g2`	`! (R) = 0`
	`sth %g2, [%g4+%g3]`	`! Store state bits`
	`ldstub [%rY], %rX`	`! Load-store unsigned byte`

Figure 23: Instrumentation hooks for SPARC prototype

overflow or *window underflow* may occur, which will cause the operating system to save or restore register values from special reserved areas on the stack. Rather than provide specialized instrumentation for these instructions, we instead execute a kernel trap to flush the user register windows to the stack at the interval boundary, and then write out the entire stack to a trace file. We discuss the details of restoring this stack trace later as part of our overview of the implementation of replay.

This stack tracing technique provides for correct replay in most cases; only programs reading from the uninitialized area of the stack reserved for register window dumps will be replayed incorrectly. These reads indicate the presence of programmer error, but our goal is to provide correct replay of all conditions, correct or otherwise, in order to aid the programmer in finding bugs. Although not currently implemented in our prototype, detection of such reads can be implemented on systems which provide kernel support for watchpoints on the stack. Solaris 2.6[2] provides support for read, write, and execute watchpoints on the stack which would allow us to implement detection of this problem without additional assembly language instrumentation.

If the user does not require complete flowback and machine query functionality, but wants to be able to replay programs correctly, we can optimize our instrumentation code for instructions which access stack variables because we know the entire stack will be traced at the interval boundary. The ALTR language allows us to easily identify such instructions by matching load and store instructions which reference addresses using the frame pointer (*%fp*) register. For load instructions, we can omit instrumentation code entirely because the entire stack will be traced. For store instructions, we need only set the machine's X bit so that the machine's flowback value will be updated at the end of the interval, resulting in a savings of one instruction per store.

4.4 Interrupts

UNIX process signals, delivered as the result of the synchronous and asynchronous exceptions described earlier, must be traced for correct replay. In order to replay an interrupt, we must be able to identify the particular instance of the execution of the instruction which was interrupted by the signal. The signal handling mechanism provided by the tracing framework allows our instrumentation library to obtain the program counter value addressing the interrupted instruction, but we also need to know the particular execution instance of this instruction: a particular instruction may be executed many times as the result of a loop or repeated function call. Mellor-Crummey and LeBlanc [12] demonstrated that instrumentation to implement a software instruction counter (SIC) can be added to a program with minimal performance penalty. A SIC is a counter which is incremented prior to every backwards branch and function call. Using the software instruction counter, which we store in reserved register *%g7*, the instruction instance interrupted by a signal can be identified by the unique pair of values *(%g7, %pc)*. We trace out the entire user context structure when a signal is delivered, including the vales of *%g7* and *%pc*, and then use this information to replay signal delivery.

Mellor-Crummey and LeBlanc's RISC implementation used a decrement and trap-on-underflow instruction on the HP Precision RISC architecture in order to trigger an exception when the appropriate SIC value is reached during replay [12]. Unfortunately no instruction exists on the SPARC to trap conditionally without testing the condition codes. Thus our instrumentation code would not only have to decrement a register and trap based on the condition codes, it would have to save and restore the condition codes themselves. On the SPARC, each of these actions can only

[2]Information based on documentation for and experience with Solaris 2.6 beta software from Sun

be accomplished via a trap into the kernel. Two traps and additional user instructions at each backward branch and function call would impose a much more serious performance overhead than we can tolerate. We propose a new implementation of the SIC for the SPARC using a virtual memory mapping trick to improve performance. Rather than counting down, we arrange for our SIC instrumentation to increment a register starting at one. While tracing an interval, we do not expect this value to ever overflow (reset back to zero). At the interval boundary, we record the final value of the SIC register for the interval, and then reset the register. During replay, we start the register at the value $0 - s + 1$ where s is the traced SIC value. Assuming the rest of replay works properly, the register will now overflow when the register has been incremented the same number of times as it was during tracing. All that remains is to devise a way to test if a register is zero or non-zero without using the condition codes, and trap if the value is zero.

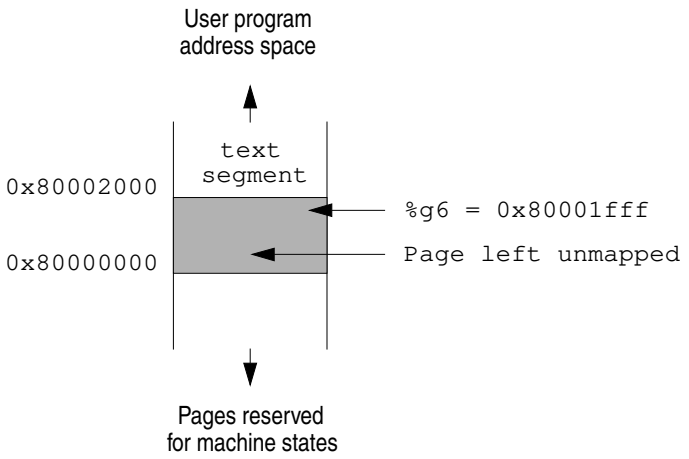

Figure 24: Virtual memory page layout just below the user program's address range

Since we already divide the address space in half to reserve space for monitoring machine storage, we can also arrange for the text segment of the user process to be loaded not exactly at virtual address 0x80000000 (the start of the top half of the address space), but instead at 0x80002000. This value was chosen because it is a page boundary on Solaris SPARC machines with 4K and 8K virtual memory page sizes. This memory layout is shown in Fig. 24. As shown earlier in Fig. 22, we reserve register *%g6* for our instrumentation and fix this register at the value 0x80001fff, exactly one byte below the start of the first page of the user program text segment. This means that the instruction `ldub [%g6 + %rX], %g0`, which loads a byte from the address specified by the sum of registers *%g6* and *%rX* and discards the result, will generate a segmentation fault that we can intercept if *%rX* contains zero, and will effectively act as a *nop* if *%rX* contains a value between one and the size of a virtual memory page [14]. Our only remaining problem is that the value in the SIC register may eventually exceed the virtual memory page size (either 4K or 8K on typical Solaris SPARC systems). Thus we need to define a function $f(x)$ where $f(0) = 0$ and $0 < f(x) < 4096$ iff $x \neq 0$. We propose two efficient implementations of this function.

4.4.1 Population Count

The function we need for our address computation can be implemented using a *population count* instruction if one is available. This instruction returns the number of one bits in a given register, and stores the result in another register. The SPARC version 9 architecture provides a `popc` instruction

with this functionality [24]. Using `popc`, we map zero to zero, and non-zero to a value between 1 and 64 (the size in bits of a register on an UltraSPARC), thus meeting the requirements of our function.

4.4.2 Exponential Bit Folding

Unfortunately not all architectures have a population count instruction. Although UltraSPARC machines implement the SPARC version 9 instruction set, other SPARC machine implement the version 8 instruction set which has no `popc` instruction, and we would like to devise a fast technique applicable to these machines as well. Exponential bit folding is a more general technique we have devised that works on machines without a population count instruction. We envision a register as a piece of paper which we repeatedly fold in half. With each fold, we logically **or** the bits on the left half of the paper with those on the right half, and the resulting sequence of half the original length is written back to the piece of paper. This technique can be implemented by repeated bitwise logical shift and **or** operations, as shown in Fig. 25.

```
srl  %rX, 16, %rY
or   %rY, %rX, %rX
srl  %rX, 8, %rY
or   %rY, %rX, %rX
srl  %rX, 4, %rY
or   %rY, %rX, %rX
srl  %rX, 2, %rY
or   %rY, %rX, %rX
srl  %rX, 1, %rY
or   %rY, %rX, %rX
```

Figure 25: Exponential bit folding implementation for 32-bit registers

At the end of the instruction sequence shown in Fig. 25, we are left with the lowest-order bit of register *%rX* set to 0 if *%rX* originally contained 0, or 1 if *%rX* originally was non-zero. In our implementation it is only necessary to fold the value down to fewer bits than are needed to represent the page size, thus only the first two folds (four instructions) are needed for our instrumentation code. Since we disregard the upper bits as we fold, a final logical **and** operation can be used to mask out only the results of the folding. These instructions do not modify the condition codes, and are very inexpensive simple logical operations.

4.4.3 Overflow

Another nice feature of our SIC implementation is that we can detect overflow. Overflow is unlikely given moderately sized intervals, but is still possible if the interval length is small and the user program makes excessive function calls or executes an extremely large number of loop iterations which exceeds the maximum 32-bit unsigned value. Since overflow will cause the SIC register value to wrap around to zero, this will cause our instrumentation code to generate a fault we can intercept. During replay, we use this fault to trigger the restore of data to replay the next interval or to transfer control to a user signal handler. During tracing, we can use this fault to report an error message indicating that SIC overflow has occurred, and suggest that the user decrease the

interval length. Another possibility is that we just end the interval when this condition is detected, and then continue tracing without having to warn the user. We only need to be concerned with SIC overflow during a single interval since we reset the SIC register at each interval boundary. Replay can only be restarted at the granularity of intervals, so the SIC value is only meaningful within an interval, and thus it is not necessary to preserve the SIC value across interval boundaries.

4.5 System Calls

In order to provide complete and correct replay of user programs, we must also trace and replay system calls, or traps into the kernel which allow the user program to interact with the operating system, external devices, or other processes. We have already made provisions in our unique-spanning read monitoring machine to handle system calls. We now discuss how we intercept these calls, and translate each one into the appropriate set of transitions on our monitoring machines.

4.5.1 System Call Effects

Whenever a system call is executed, we need to determine the precise size and location of each region of memory in the user process written by the system call. Unfortunately, for our UNIX prototype, there is no systematic way to determine this information at run-time. The information is different for each system call, and can only be obtained by examining the corresponding C function prototype, manual page, or operating system source code. This is one of the primary limitations of our tool: in order to correctly support system calls, we must create a version of our instrumentation library which is very tightly coupled to the implementation details of a particular version of a particular operating system. Experience has shown that even changes in minor releases in an operating system may introduce new system calls. Still, operating system developers are tuned in to the needs of tool developers, especially because advanced debugging tools benefit everyone who develops for a particular operating system. Advanced operating system support for debugging, such as the `/proc` filesystem [4], already exists in several UNIX operating system variants. The Solaris `/proc` implementation provides the ability to trace the entry to and exit from system call traps. Perhaps in the future support could be added to obtain additional system call information such as affected memory regions. Another possibility is that we could trace entry to each system call, and then use the `/proc` watchpoint facility to dynamically determine affected memory regions.

Since operating system supported watchpoints were not available at the time of the development of our prototype, we developed our own approach for intercepting and tracing system calls. Our basic goal was to create a wrapper function for each system call in our instrumentation library, which would then be *interposed* [21] in front of the standard C library system call routines. The wrapper executes the appropriate system call trap, writes a trace record indicating which system call was executed and its return value, and then invokes another routine to perform appropriate machine transitions on affected memory regions. To simplify development of these wrapper routines, we created an extended C-like prototyping syntax for system call functions, and a tool to automatically generate the C code for the wrapper functions from the prototypes. An example prototype for the `read(2)` system call is shown in Fig. 26. The prototype is used to generate the C language prototype, and the special *WRITES* { } clause is used to indicate that the system call causes the operating system to write to the user address space at the address specified by the `buf` parameter a number of bytes which is given by the return value of the system call. This prototyping method handles the vast majority of UNIX system calls. Unfortunately, there are some special cases such as

`readv(2)` and `ioctl(2)` that do not fit nicely into this model. We have added special syntax and some hand coding to deal with these few exceptions.

```
ssize_t read(int fd, WRITES{ void *buf = retval } , size_t nbyte);
```

Figure 26: System call generator prototype for `read(2)`

We need to explicitly trace the system call return value, stored in register *%o0* on return from a system call trap on the SPARC, because this value may be immediately referenced in the register upon return from the system call, and may affect the control flow of the program. Following the actual trap, the wrapper routine calls the SysTrace routine to trace the trap information and return value. At this time, the trap is assigned a system call id number which is unique to the interval. Pseudo-code for this routine is shown in Fig. 27.

---

ushort_t SysTrace(int *code*, int *subcode*, void **retval*)

```
1  if (system_call_id == 0)
2      fatalerr("System call id overflow");

3  trace_system_call(system_call_id, code, subcode, retval);

4  if (retval == (void *) -1) {
5      set_machine_state(&errno, Syscall);
6      set_flowback_value(&errno, system_call_id);
7  }

8  return system_call_id++;
```

Figure 27: SysTrace algorithm

After tracing the system call trap, we need to perform a system call transition on the machine associated with the address of the `errno` word if the system call failed. We also use this routine to detect overflow in our system call counter. Assuming the excessive number of system calls was not a bug itself, the user can decrease the interval length to avoid this type of overflow. When performing system call transitions on machines, we also store the system call id number into the flowback value field of each monitoring machine. This provides more powerful flowback data for addresses written by system calls. We can store this value in place of an interval number because the *Syscall* and *Read After Syscall* states already imply that the value was written by a system call in the current interval, whose number is known at run-time. For these data values, we can now report that the value was written in the current interval, and we can report the precise system call trap code and return value for the call by correlating the id number to our system call trap trace data. During replay, our stub does not re-execute the system call. Instead it restores the trace of the affected memory regions and returns the traced system call return value. We discuss the specifics of restoring system call trace data later in our section on the implementation of replay. We now turn to how addresses written by system calls are traced by the wrapper routine.

4.5.2 System Call Tracing

If the system call was successful, the wrapper routine then builds an array of structures identifying the starting address and length in bytes of each contiguous region of memory written by the system call. This vector is then passed to a generic system call tracing routine implemented in the instrumentation library. Pseudo-code for this routine is shown in Fig. 28.

```
void SysWrite(ushort_t id, iovec_t *iov, int count)

1   for each word-aligned addr in regions iov[0] ... iov[count - 1] {
2       if ((machine_state(addr) & (RWS)) == 000)
3           trace_in_child(addr);

4       syscall_transition(addr, id);

5       if (addr > stack_pointer)
6           trace_in_parent(addr);
7       else
8           send_to_child(addr);
9   }
```

Figure 28: SysWrite algorithm

We use an array of `iovec_t` structures, each containing a base address and length field, to describe the regions of memory written by the system call. The goal of this routine is to perform the system call transition on the monitoring machines associated with each affected address, and to transfer the values written by the system call to the controlled child process; they will be traced by the child if the monitoring machine is in the *Read After Syscall* state at the end of the interval. There are a number of special cases we need to handle first. The first **if** clause identifies machines who have the R, W, and S bits all zeroed. This pattern identifies the states *Read First* and *Read and Written*. If the machine is already in either of these states, then the value at the corresponding address in the child process has already been the subject of a unique-spanning read, and thus needs to be traced. So before we overwrite this value with the new value written by the system call, we direct the child to write out the current value to a trace file.

Next we perform the system call transition on the monitoring machine. The *id* parameter is a 16-bit identifier value for the system call unique to the current interval. This value is obtained from the result of the call to the SysTrace routine. When we perform a system call transition, we store this 16-bit value in the low-order 16 bits of the state machine word. We had previously indicated that this space would be used to store an interval number to support flowback debugging queries. However, if a machine is in the *Syscall* or *Read After Syscall* states, we know it has been written in the current interval by a system call. Thus instead of storing just the current interval number, we store the system call identifier. We later replace this with the interval number during the interval boundary code.

Finally, before we transfer the data to the child process, we make one final check to see if this address is in the stack. If it is, we immediately trace the data in the parent process rather than

transfer it to the child. We are forced to do this because the stack in the parent may have changed since we forked the child, and thus writing system call data on the stack to the corresponding stack addresses in the child process might overwrite the child's current stack frame. Otherwise, since all of the child's state is on the stack or in our library's private data, we are free to write to the child's address space to buffer our system call data. The actual code for the routine is quite different from the pseudo-code shown in Fig. 28 because rather than tracing and sending messages inside the loop, we transfer addresses to the child using a shared memory buffer to minimize copying, and we perform the transfers in batches to minimize context switches between the parent and child waiting for messages to be processed.

4.6 Interval Boundaries

As discussed earlier, we trigger interval boundaries using the UNIX interval timer facility. We allow the user to specify the interval length as a number of virtual seconds of CPU time. Our instrumentation library installs a callback for the timer signal with the RDB framework, and then executes an interval boundary routine in the signal handler. Inside the interval boundary code, we need to examine each machine, trace the corresponding address and value depending on the machine's state, and then perform the interval boundary transition on the machine. Once this is complete, execution resumes at the interrupted instruction. We now examine several implementation details of the interval boundary algorithm.

4.6.1 Atomicity of Instrumentation

Since our interval boundary algorithm will examine each monitoring machine and make tracing decisions based on the machine's state, it is critical that our sequences of instrumented code be *atomic* with respect to interval boundaries. At the simplest level, we can imagine that an instrumented sequence of assembly code to perform a read from address A consists of two clauses: `read A` and `read-transition A`. Since the interval boundary timer can trigger at any user instruction, it may very well trigger after A has been read, but before the corresponding machine transition has occurred. In this case, A may have just been read by a unique-spanning read, but we will not know that it needs to be traced without performing the corresponding machine transition.

The precise placement of interval boundaries is not important, so it is acceptable to simply delay the execution of the interval boundary code in this case until after the corresponding read transition, assuming that we can detect this condition. We use the RDB *Instrumentation Offset Table* described earlier to compare the interrupted program counter to the known sequences of instrumented code. If the signal is delivered with the *%pc* in such a sequence, we then set a breakpoint in the process just after the instrumentation sequence and continue. The signal handling mechanism provided by RDB allows us to intercept the subsequent breakpoint fault, and then execute the normal interval boundary code with the new interrupted context.

4.6.2 Interval Boundary Algorithm

We have already discussed several issues related to the interval boundary algorithm. Our basic goal is to examine the state of each monitoring machine to determine if the corresponding address and its value in the controlled child process need to be traced. Following this decision, we need to reset

the state of each machine back to the *Unaccessed* state. Additionally, if the value was modified in the current interval, we want to update the flowback value field in the machine state word with the number of the interval which just ended. The complete algorithm is more complex, and we discuss it in detail in four parts.

```
void IntervalBoundary(ucontext_t *ctx)

1   flush_register_windows();
2   trace_stack(ctx);
3   trace_context(ctx);
4   flush_trace(syscall_trap_trace);
4   flush_trace(syscall_addr_trace);
5   flush_trace(syscall_flow_trace);
6   child_set_brk(sbrk(0));
```

Figure 29: IntervalBoundary algorithm, Part I

Upon entry to the interval boundary routine, we flush the SPARC register windows to the stack and then write out the trace of the user stack. We also trace the complete interrupted machine context as well. This contains the values of all registers, and also the various status registers which include the integer and floating-point condition codes. We need to be able to restore these values for correct replay. By tracing out the context, we can restore the complete set of condition codes and registers by reading back the traced context and performing a `setcontext(2)`. Next we flush each of the trace file buffers associated with system call trace data to write out any remaining buffered data from system calls during the interval. Finally, we arrange for the controlled child process to synchronize its break with the parent. This is important because we need to send over addresses for the child to trace. Some of these addresses may be in a range of the heap which was mapped in during the interval because the user program dynamically allocated storage. We synchronize the break values so that the child can correctly reference the initial values of these addresses as zero, as new heap pages are zero-filled.

```
void IntervalBoundary(ucontext_t *ctx)

 7  for each word-aligned addr in mirror pages except for stack mirror pages {
 8      if ((X) ≠ 0)
 9          set_flowback_value(addr, intvl_num);
10      if ((RW) == 00)
11          send_to_child(addr | ((S) ≫ S_SHIFT));
12      set_machine_state(addr, Unaccessed);
13  }
14  child_start_tracing();
```

Figure 30: IntervalBoundary algorithm, Part II

Once the stack and register context have been traced, we can begin examining the state of each monitoring machine which does not correspond to an address in the stack to determine whether to trace its corresponding address and value. As discussed earlier, this tracing will be done using the values in the child process's address space. In addition, if the address was modified during the current interval, the monitoring machine in the child's address space contains the flowback value for the previous modification, which is what we want to trace along with the value. The next part of the algorithm executed in the parent process is shown in Fig. 30. We first identify states where the value was modified, and update the flowback value. Then we identify states requiring tracing, and send the corresponding address to the child. For performance reasons, this is not implemented as a message per machine, but instead each send just stores the address in a shared memory buffer for batch transfer. Following the loop, a message is sent the child indicating it can begin tracing.

In addition to tracing addresses and values corresponding to monitoring machines, it is crucial that we be able to distinguish between data traced as the result of a unique-spanning read following an interval boundary, and data traced as the result of a unique-spanning read following a system call. In the first case, we need to restore the traced data into the process's address space prior to the restart of the interval for replay. In the second case, we need to restore the traced data on-the-fly upon entry to our system call wrapper function during replay. In order to simplify the management of these two different data types, we maintain two separate trace streams of *(address, value, flowback-value)* tuples: one for 'standard' trace data and another for system call trace data. Since our machines correspond to word-aligned addresses and a word on the SPARC is four bytes, we know that the lower two bits of each address we send to the child will both be zero. We take advantage of one of these unused bits to store a copy of the machine's S bit, as shown in Line 11 of Fig. 30. When the child process receives a batch of addresses, it examines this bit to determine whether the address should be traced to the standard or system call trace stream.

```
void IntervalBoundary(ucontext_t *ctx)

15  for each word-aligned addr in mirror pages for stack {
16      if ((X) ≠ 0)
17          set_flowback_value(addr, intvl_num);
18      set_machine_state(addr, Unaccessed);
19  }
```

Figure 31: IntervalBoundary algorithm, Part III

For monitoring machines which correspond to addresses in the stack, we can perform a simpler loop which simply checks the X bit to see if the flowback value should be updated with the current interval number, and then resets the machine state back to *Unaccessed*. Since we have already told the child to begin tracing, this second loop shown in Fig. 31 proceeds concurrently with the child's tracing activity.

The end of the interval boundary routine is shown in pseudo-code in Fig. 32. We need to record a marker in each trace file indicating the end of an interval. We then output an interval summary record indicating the ending software instruction counter value for the interval, the number of system call traps which were executed, and the approximate virtual CPU time which elapsed during the interval. We also need to record the current virtual memory mappings and their page

```
void IntervalBoundary(ucontext_t *ctx)

20  trace_interval_boundary();
21  trace_memory_mappings();
22  trace_file_offsets();
23  fork_child_process();
24  set_sic_register(0);
25  system_call_id = 1;
```

Figure 32: IntervalBoundary algorithm, Part IV

protections; they will be restored prior to replay of the next interval. To facilitate rapid incremental replay, we also record the file offset of each of the other trace files in a special trace file, to allow us to jump directly to the start of the trace data for a particular interval. Finally, we fork a new child process to buffer values for the next interval, and reset the SIC register and system call trap counter.

5 Replay Prototype Implementation

In this section we discuss theory and implementation of execution replay based on our implementation of a tracing engine for unique-spanning$_0$ reads described thus far. Our work is designed to allow the debugger to control replay at the granularity of intervals, by replaying a sequence of intervals in order, or by allowing the user to re-execute portions of intervals in any order, potentially guided by flowback information from the monitoring machines. Our goal here was to implement a *mechanism* for controlling replay but not a set of *policies* restricting how the debugger should operate. Later we discuss how the replay back-end can be combined with a debugger interface library to allow replay functionality to be combined with existing debugging tools, or to provide the foundation for new types of debugging tools.

5.1 Program Instrumentation

The first issue we considered in designing the replay engine was whether or not the executable that runs during replay should in effect be the same program that was compiled and instrumented to perform tracing. One issue we immediately must consider is the size of the program text segment and the locations of various data in memory. Since our trace files consist of traced addresses and values from the original run, we need the replay process's address space to be laid out precisely as it was during tracing in order to provide correct replay. If we tried to re-execute an uninstrumented executable, the functions would appear at different addresses since no instrumentation code would be present, and thus replay would not work.

Although we have made every effort to optimize our instrumentation code for tracing, it still adds overhead to the program run, so another possibility is that we could reduce the performance overhead for replay by modifying the instrumentation. We could produce a different instrumented executable for use in replay which is identical to the tracing executable, except that the tracing

instrumentation instructions could be overwritten with *no-ops* or with a different type of instrumentation code of identical size which provides some other type of information for replay.

There were several reasons why we decided not to pursue the idea of multiple executables. The primary reason was that the instrumentation code which performs tracing provides useful information which the debugger can access during tracing or during replay. Our eventual goal is to have replay taking place under the control of a debugger which provides standard facilities such as breakpoints and data display capabilities to the programmer. We realized that maintaining a meaningful machine state corresponding to each data word in the user program provides another source of information the debugger can use to answer queries. By allowing the instrumentation code to re-execute as part of replaying an interval, we can allow the programmer to stop the target program in the debugger, and then obtain the machine state from the debugger. The instrumentation enables the debugger to answer queries of the form "Has this variable been modified yet in this interval?", "Has this variable been written by a system call in this interval?", or "Has any instruction accessed this value yet in this interval?" The instrumentation to update the software instruction counter is needed for replay to trigger the replay of interrupts, and overall we know that the overhead of the instrumentation code will be no worse during replay than it was during tracing, and we already deemed that acceptable. So we produce a single executable for both tracing and replay, but provide code to handle each mode differently in the instrumentation library.

5.2 Restoring the Address Space

Before we can restore traced values into the process address space, we need to first restore the state of the address space to its state prior to the execution of the desired interval. When we refer to the *state* of the address space we are talking about the set of valid address ranges and associated virtual memory protections. Depending on the operating system, these ranges and permissions may be defined and implemented in a variety of ways. Since our prototype is for Solaris, we discuss the details of our implementation for this platform, but the ideas are applicable to other modern operating systems as well.

The Solaris virtual memory system is page-based, and a process's address space consists of a set of *mappings* onto the address spaces of objects in the system's virtual memory [9]. The Solaris `/proc` filesystem [4] provides a facility for obtaining the set of mappings currently associated with a process address space, including the virtual address ranges and virtual memory page protections. During tracing, the instrumentation library uses this facility to record the set of mappings prior to the execution of each interval. We also explicitly record the break (the address marking the extent of the heap) and the contents of the stack, as discussed earlier.

Besides the stack and heap pages, and pages storing the text and data of the program and shared libraries, a Solaris process may establish other mappings such as shared memory segments, or mappings to files, devices, or anonymous memory pages using `mmap(2)`. The process may also change virtual page protections using `mprotect(2)`. Thus in addition to restoring the address space, we must also provide special interposed versions of these system call functions which actually perform the system call trap again during replay to make the appropriate changes to the address space. In order to set up the address space properly prior to the start of the interval, we walk through the set of traced mappings and compare them to the current set of mappings in the process. If a mapping exists currently but does not exist in the trace records, we remove it. Conversely, if a mapping was traced but does not currently exist we add it by mapping the appropriate range of

pages from the `/dev/zero` device, which provides a source of anonymous pages. Since we take care of restoring all accessed values, anonymous pages can be used during replay in place of mappings of devices and files from the original execution. We also reset protections on existing protections if they do not match the traced protections.

5.3 Restoring the Stack

When an interval boundary is reached, we traced the interrupted context and the contents of all stack pages to a trace file. We must make special provisions for restoring this information because our instrumentation library code is itself running on the user program stack, and thus overwriting the stack with traced data may prove fatal to our library code. To solve this problem we allocate another stack elsewhere in the process's address space by mapping a small number of pages of anonymous memory from `/dev/zero`. We can then create a `ucontext` structure which has the stack pointer register referencing the top of this alternate stack and the program counter register referencing the address of our stack restore routine, and then execute the `setcontext(2)` system call to transfer control to this routine on the temporary stack. We can now safely overwrite the user stack pages, load the context which was originally interrupted by interval boundary, flush the register windows, and then execute another `setcontext(2)` to jump back into the user program.

5.4 Restoring Trace Data

The trace data accumulated during interval boundaries and system call wrappers is written out to two separate trace streams, one for 'standard' trace data and one for system call trace data. Each trace stream stores a sequence of (*virtual address*, *value*, *flowback value*) tuples that we need to restore into the user address space during replay. A special tuple is used to mark the end of the data associated with a given interval. We now briefly discuss how the standard trace data is restored to the user address space, and then discuss how system calls are replayed.

The interval handler for replay first resets the state of each monitoring machine back to the *Unaccessed* state as during tracing. Then we read in the data from the standard trace stream, restoring each value to the appropriate virtual address, and storing the traced flowback value in the corresponding machine state word. The standard trace data stream consists of data traced following unique-spanning$_0$ reads, the first read from an address since the previous interval boundary. Regardless of the order of the traced values, each value was at the given address at the start of the interval during tracing. Thus we can restore all of this trace data to the user address space prior to the replay of the interval without considering the order of trace records.

5.5 Replaying System Calls

In order to replay a system call, we cannot restore the traced value to the appropriate virtual address and update the flowback value until the user program calls the system call wrapper function in the instrumentation library. Upon entry to a wrapper, we consult the system call trap trace file to determine the return value for the system call, and then restore the traced data and flowback values, and perform the system call transition on the monitoring machine associated with each address for which trace data was restored. The wrapper function is aware of what system call trap is being replayed so it can perform special handling for some system call traps, such as actually

re-executing `mprotect(2)` calls. Another feature we provide in the wrapper is the ability to re-execute `write(2)` system calls to the file descriptors associated with `stdout` and `stderr` so that the user is provided with some visual feedback that replay is working properly.

When restoring standard trace data, we did not need to be concerned with the order of trace data since all of the standard trace data for an interval needs to be restored prior to the replay of the interval. When restoring system call data, we need the data in the trace file to be ordered; we want to avoid scanning the entire set of traced data for the interval at each call site. Unfortunately, because of the design of the tracing machine, the system call trace data can be completely unordered. Consider the sequence of program events shown in Fig. 33:

1 System Call Write A
2 System Call Write B
3 Read B
4 System Call Write B
5 Read A
6 Interval Boundary

Figure 33: Sequence of program events leading to unordered system call trace data

The problem is that system call event 4 overwrites the value stored at address B, which needs to be traced because this value was read following system call event 2. As a result, the value stored at address B must be immediately written to the trace file as part of the handling for system call event 4. Meanwhile, address A was previously written by system call event 1, but is not read until event 5, and will not be traced until the interval boundary. We need to restore the value traced from address A first during replay, but this data comes later in the trace file. This example illustrates that in order to provide for efficient system call replay, we need to sort the system call trace data by system call id number prior to replaying the program. The data can be sorted using external merge-sort or other algorithms, and once sorted, can be written back to the trace file and used for any number of subsequent replays. We feel that since this cost is amortized over all subsequent replays of the program, the sorting requirement does not impose an unreasonable performance penalty. We added code to our debugging library, discussed in the next section, to automatically sort trace system call trace files when opened by the debugger, if not already sorted, and write the sorted data back to the trace file for use in subsequent replays.

5.6 Interval Boundaries

We must be able to detect the precise execution instance of the instruction which marks the end of each interval during replay so that we can jump back into our interval handler code and set up for replay of the next interval. The instruction instance that was interrupted by the interval boundary timer signal during tracing was recorded by tracing the current (*SIC*, *%pc*) pair in the interval boundary handler. Whereas a traditional debugger breakpoint consists only of a program counter value, we now need to set a breakpoint which only fires at a particular (*SIC*, *%pc*) pair. We can do this with very low overhead using our existing SIC instrumentation, which causes a segmentation fault to occur when the SIC register overflows. Prior to the start of the interval, we set the SIC register to the value $0 - s + 1$ where s is the traced SIC value. This will cause a fault to occur when the SIC has been incremented the same number of times as it was during the original program run. In the fault handler, we then set a standard breakpoint at the *%pc* value associated

with the interval boundary and continue the program. When this breakpoint fires, we then enter the interval boundary code and set up for the replay of the next interval.

5.7 Replaying Interrupts

We can replay asynchronous interrupts in the same manner as interval boundaries. We trace the (*SIC*, *%pc*) pair for each asynchronous interrupt, as well as the action taken in response to the signal. Since we have only a single SIC register to use to generate the faults to trigger our (*SIC*, *%pc*) breakpoints, we keep a queue of these events and at each fault load the appropriate value to trigger the next interrupt or to trigger the interval boundary handler. We also provide the facility to set these type of breakpoints to the debugger, as described in the next section. This feature enables the programmer to use the debugger to set *replay breakpoints*, where a particular point in the program's execution can be recorded, and then replay can be restored back to that point.

5.8 Incremental Replay

To support incremental replay, we want to allow the debugger to stop the process during replay at any time, and then direct the instrumentation library to set up for replay of an arbitrary interval. We can support this functionality by providing a routine similar to the replay interval handler that restores the address space, stack, standard trace data, and initializes the SIC register for the next interrupt breakpoint. Since we trace the byte offsets of all trace files prior to the start of each interval during the original execution, this function need only seek each trace file to the appropriate position in order to prepare to restore the data for any interval. The instrumentation library makes the address of this function known to the debugger back-end, described following the performance results section, so that the debugger can invoke it any point during replay.

6 Performance Results

To assess performance, we ran several experiments to measure run-time slowdown and rate of trace generation. Reasonable slowdowns are important, but a low trace rate is crucial. Techniques that require tracing megabytes of data per second are doomed to fail. We ran our tracing engine on the five programs shown in the table below, using an interval length of 20 seconds. The results are shown in Fig. 34. *life* performed 20,000 iterations of a life simulation on an 100x100 world. *mesh* performed 1000 iterations over a 1000x1000 grid to solve a simple differential equation. *go* played the game of go against itself on a 27x27 board. *m88ksim* simulated the Motorola 88100 microprocessor running the Dhrystone benchmark. These last two programs are from the SPEC95 integer suite. *cc1* is the compilation pass of the GNU C compiler run on the gcc source file cp-decl.c (which contains 9,169 lines and is 307 kilobytes in size). All programs were compiled without optimization and were run on a two-processor 200MHz Sun UltraSPARC running Solaris 2.5.1. Traces were written to the local SCSI disk.

The table shows the uninstrumented running time of each program run, the instrumented running time (and slowdown) measured as the wall-clock time to completion, the total trace size, the size of the stack trace file and its percentage of the total data traced, and the rate of trace generation. The run-time slowdowns are typically around 2.5–3.5. Considering the functionality

	life	*mesh*	*go*	*m88ksim*	*cc1*
Running time	15m48.567s	22m16.695s	19m34.260s	18m29.160s	0m25.696s
Instrumented time	42m25.801s	58m57.469s	1h03m05.005s	1h08m23.858s	1m18.146s
Slowdown	2.68	2.65	3.22	3.70	3.04
Total trace size (kb)	6975	752651	70799	39289	12715
Stack trace size (kb)	721 (10%)	697 (<1%)	1263 (2%)	3900 (10%)	3460 (27%)
Trace rate (kb/sec)	2.7	212.8	18.7	9.6	162.7

Figure 34: Overheads of replay tracing

provided by the traces, we consider this to be an acceptable slowdown for many programs, and hope to improve our performance in the future. The 20-second interval length means that the traces are sufficient to re-execute *any* 20-second interval of these long runs, and the time required to set up the state for a replay is negligible (as each interval's trace is contiguous in the trace file). Also note that our tracing engine is fully functional: all system calls and libraries are instrumented for these runs.

The trace rates are worth examining more closely. For *life*, *go*, and *m88ksim*, the 2.7–18.7 kilobytes/sec of trace generation is practically negligible. Runs of over a day's length can be accommodated with a gigabyte disk. *mesh* produces a high trace rate because its locality of reference is the worst-case for unique-spanning read tracing: in each iteration through the grid, every element is set to the average of its neighboring elements, meaning that in each interval there is a read of every element not preceded by a write (which must be traced). The reason that *cc1* produced a trace rate of 162.7 kilobytes/sec is unclear and is being investigated. Perhaps the symbol table or control-flow graph computed by the compiler tends to be written once and then read many times, making most of the reads unique-spanning. In short, some variable access patterns tend to increase overhead, but even mesh (which is our proposed worst-case scenario) did not show significant slowdown.

We performed several runs of *cc1* to get a feeling for how the trace rate varies with interval size. For a 5, 10, or 15 second interval length, the trace rate only increased from 184 to 200 to 230 kilobytes/sec, respectively. These higher trace rates might well be acceptable in many cases, especially considering that a 5-second interval length would allow any replay request to complete almost immediately, letting long compilation runs be debugged without replaying from the start. The run-time slowdown for *cc1* increased to only 3.2 for 5-second intervals (from a slowdown of 3.0 for 20-second intervals).

In summary, traced programs took 2.5–3.5 times as long to complete. This probe effect *is* a non-trivial cost, but trace rates are low enough to support long runs without making tracing a bottleneck. This trace-and-replay technique works well for moderately long program runs (5–60 minutes), thereby filling a useful debugging niche. If the slowdown can be tolerated, the payoff is substantial: once a trace is saved, *any* part of a run can be reproduced quickly regardless of its length. We hope to do more detailed performance analysis of the prototype in the future, particularly in the area of interaction with the virtual memory system and with memory caches.

7 Debugger Back-End

Our prototype replay implementation was designed to allow the user to replay programs from start to finish without the aid of a debugger, or to take advantage of the replay engine to perform more powerful queries using replay in the debugger. To this end, we developed a debugger back-end library which allows existing debuggers such as GDB [20] to easily incorporate replay functionality and also provides a framework for building a new debugger specifically designed to take advantage of incremental replay. The design of a typical debugger can be factored out into several discrete components: code to control another process and obtain or manipulate its state, code to process symbol table information and provide mappings between program source code and process state, and code to display information to the user and allow the user to issue queries.

The process control code for a UNIX debugger is typically implemented using a set of operating system primitives to allow one process to control another, such as the Solaris `/proc` filesystem [4] or the `ptrace(2)` system call provided in some other systems. Our goal was to provide a simple but powerful programming interface which can be used in place of the process control layer provided by the operating system. This design idea is very similar to the idea of a machine-dependent 'nub' for debugger process control used in the CDB debugger [10]. However, as we shall see, our back-end library needs to provide additional functionality beyond a unified interface to the operating system's process control mechanism. Our requirements for the back-end were as follows:

- The library should provide the same basic facilities as the operating system interface for controlling processes, and should be easily pluggable into existing debugging systems. We used the requirements of the GDB `target_ops` structure [7] as a measure of the requirements of a full-featured debugger. GDB uses this structure to store a vector of pointers to functions which implement various process control primitives in terms of the process control mechanism provided by a particular operating system variant. It should therefore be straightforward to implement a new `target_ops` structure in terms of the RDB back-end.

- The library should be able to interoperate with different RDB instrumentation libraries. We have experimented with different machines, machine state storage methods, and instrumentation code, and would like to continue to do so in the future. An important requirement is that if we implement a new instrumentation library for trace-and-replay, this should not require implementation changes in the back-end. Furthermore, the back-end should be able to support programs linked to different instrumentation libraries *simultaneously*.

- The library must not restrict the design of the debugger. It should support the control of multiple processes at the same time so that the debugger can support this, and it must allow the debugger to register callbacks to occur during the initialization of a controlled process so that the debugger can redirect program output to different terminals, or to a graphical display.

- The library must effectively mask the presence of the RDB run-time framework and the instrumentation library. If the debugger queries the back-end to determine what signal handlers are installed by the user process, the back-end must decode the user signal handler array inside of the framework rather than reporting the signal handler registered with the operating system, which is the handler inside of the RDB run-time engine. If the process stops on an event of interest which is detected inside of a routine in the run-time engine, the back-end must conceal the presence of the framework routines so that the debugger can report a stack

back-trace which shows that the program is stopped at the interrupted location in user code, **not** that the program is stopped inside a routine inside the RDB framework library.

- The library must not impose a significant performance overhead beyond that already incurred by the operating system's process control facilities. In particular, excessive context switching between the debugger and the controlled target process is extremely undesirable.

7.1 Programming Interface

The complete application programming interface for the back-end library breaks down into several categories of functionality. We present the actual API C function prototypes associated with each category to illustrate the simplicity and flexibility of the library.

- *Process Control* The back-end allows the debugger to create new controlled processes for either tracing or replay, set a variety of options for tracing and replay, such as the length of an interval. The back-end also has functions to stop and continue processes and to wait for the process to stop on an event of interest, such as a breakpoint or signal.

```
int rdb_proc_trace(rdb_proc_t *pr, const char *shell, const char *path,
    const char *args, const char *envp[ ], const rdb_propts_t *opts);

int rdb_proc_replay(rdb_proc_t *pr, const char *path, rdb_trset_t *trace,
    const rdb_propts_t *opts);

int rdb_proc_restore(rdb_proc_t *pr, rdb_intvl_t *intvl);
int rdb_proc_destroy(rdb_proc_t *pr);
int rdb_proc_run(rdb_proc_t *pr);
int rdb_proc_wait(rdb_proc_t *pr, prstatus_t *status, ulong_t *flags);
int rdb_proc_stop(rdb_proc_t *pr);
int rdb_proc_getopts(rdb_proc_t *pr, rdb_propts_t *opts);
int rdb_proc_setopts(rdb_proc_t *pr, const rdb_propts_t *opts);
```

- *Process Signals* The debugger can use the back-end to control a set of traced signals. Each traced signal causes the process to stop, and then the debugger can report that a signal is about to be delivered. There are also functions to deliver and clear pending signals, and to obtain the set of signal handlers associated with the process.

```
int rdb_sig_add(rdb_proc_t *pr, int signo);
int rdb_sig_delete(rdb_proc_t *pr, int signo);
int rdb_sig_set(rdb_proc_t *pr, const siginfo_t *info);
int rdb_sig_acts(rdb_proc_t *pr, struct sigaction *act);
int rdb_sig_trace_get(rdb_proc_t *pr, sigset_t *set);
int rdb_sig_trace_set(rdb_proc_t *pr, const sigset_t *set);
int rdb_sig_max(rdb_proc_t *pr);
```

- *Address Space Manipulation* The back-end allows the debugger to read and write the target process's address space, and to obtain the current set of virtual address mappings which describe the set of valid virtual address ranges in the process. We also provide a function that allows the debugger to obtain the current monitoring machine state and flowback values for a range of addresses in the target.

  int rdb_mem_read(rdb_proc_t *_pr_, rdb_addr_t _addr_, void *_buf_, size_t _nbytes_);
  int rdb_mem_write(rdb_proc_t *_pr_, rdb_addr_t _addr_, **const** void *_buf_, size_t _nbytes_);
  int rdb_mem_maps(rdb_proc_t *_pr_, prmap_t *_maps_);
  int rdb_mem_nmaps(rdb_proc_t *_pr_);

  int rdb_mem_state(rdb_proc_t *_pr_, rdb_addr_t _addr_, rdb_state_t *_states_,
  ulong_t *_fbvals_, size_t _nwords_);

- *Machine Register Manipulation* The library provides functions for the debugger to get and set the contents of machine registers associated with the process. A function is also provided to obtain the contents of the SIC register.

  int rdb_greg_get(rdb_proc_t *_pr_, prgregset_t *_gregs_);
  int rdb_greg_set(rdb_proc_t *_pr_, **const** prgregset_t *_gregs_);
  int rdb_fpreg_get(rdb_proc_t *_pr_, prfpregset_t *_fpregs_);
  int rdb_fpreg_set(rdb_proc_t *_pr_, **const** prfpregset_t *_fpregs_);
  int rdb_xreg_get(rdb_proc_t *_pr_, prxregset_t *_xregs_);
  int rdb_xreg_set(rdb_proc_t *_pr_, **const** prxregset_t *_xregs_);
  int rdb_sic_get(rdb_proc_t *_pr_, ulong_t *_sic_);

- *Trace File Management* The debugger can use the back-end to open sets of trace files and obtain information about a traced run. The debugger can determine how many intervals were traced, how much virtual CPU time was spent in each interval, how many system call traps were executed in each interval, and the trap code, subcode, and return value of each system call executed during an interval.

  int rdb_trset_create(rdb_trset_t *_trace_, **const** char *_basename_, pid_t _id_);
  int rdb_trset_destroy(rdb_trset_t *_trace_);
  int rdb_trset_info(rdb_trset_t *_trace_, rdb_trinfo_t *_info_);
  int rdb_trset_truss(rdb_trset_t *_trace_, rdb_intvl_t *_intvl_, rdb_trap_t *_traps_);

- *Breakpoints* The back-end provides functions to allow the debugger to set either standard breakpoints at a particular *%pc* value or to set a *replay breakpoint* by specifying a (*SIC*, *%pc*) pair. Each type of breakpoint is registered as an event of interest which causes the process to stop.

  int rdb_brkpt_create(rdb_brkpt_t *_bp_, rdb_proc_t *_pr_, rdb_addr_t, _addr_, ulong_t _sic_);
  int rdb_brkpt_set(rdb_brkpt_t *_bp_);
  int rdb_brkpt_clear(rdb_brkpt_t *_bp_);
  int rdb_brkpt_destroy(rdb_brkpt_t *_bp_);

- *Error Handling* When any of the library functions report an error condition, the debugger can use the error handling functions to obtain the current error code and convert it to a meaningful text string.

  const char *rdb_errmsg(int *err*);
  int rdb_errno();

These categories cover most of the important functionality required by a source-level UNIX debugger. There is some other functionality we hope to add in the future, including the ability to trace the execution of system calls in much the same way the API supports tracing signals. We discuss this and other enhancements in the future work section at the end of this document.

7.2 Debugger Agent

One of our goals for the back-end was that it must be able to support different instrumentation methods and instrumentation libraries simultaneously, and that developing new instrumentation libraries should not require modification of the back-end. To meet this goal, the RDB run-time engine provides facilities for the instrumentation library to communicate with the back-end via a separate agent thread which lives inside of the target process. The agent is implemented as a separate light-weight process [18] (LWP) in the user process, and is conceptually based on the *agent LWP* support provided by the Solaris 2.6 `/proc` filesystem[3]. The agent allows the debugger to effectively execute code in the instrumentation library on behalf of the debugger when the user program LWP is performing other actions or stopped on an event of interest. By implementing routines in the instrumentation library that allow the debugger to obtain information such as machine state and flowback values, we can insulate the back-end from the different storage techniques and algorithms used by different instrumentation libraries. The debugger back-end and the agent communicate using `/proc` and a simple message protocol. Fig. 35 shows a block diagram of the complete RDB architecture.

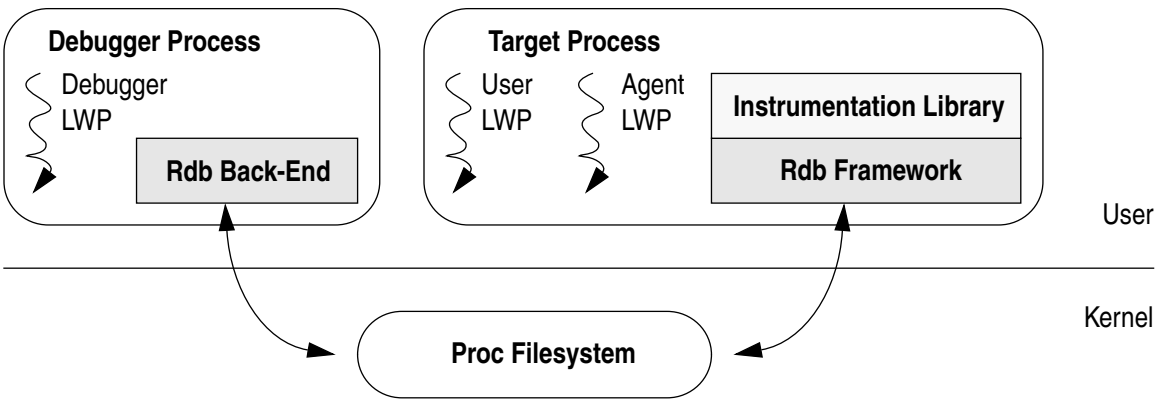

Figure 35: Debugger architecture including agent LWP

[3]Information based on documentation for and experience with Solaris 2.6 beta software from Sun

Each instrumentation library provides processing routines for the agent LWP to execute for the following debugger commands:

1. Update instrumentation library based on changes made to the current set of run-time options made by the debugger. Run-time options include settings such as the interval length.
2. Create, enable, disable, or destroy replay breakpoints.
3. Convert specified vectors of user program addresses to vectors of the corresponding addresses of monitoring machines or flowback values.
4. Seek trace files to specified locations in order to set up for replay of a different program interval.

7.3 Signal Handling

The issue of tracing the delivery of UNIX process signals to the target in the debugger raised a potentially serious performance problem in the design of the back-end. The back-end API offers a set of functions to stop the process when one of a specified set of signals is delivered to the user program. This feature is used by UNIX debuggers such as GDB to allow debugging of signal handling code, or to report the location and stack trace of synchronous exceptions such as segmentation faults. During the execution of a potentially buggy user program, some signals may be generated which the debugger has asked to trace but are not associated with the user program; they are associated with the instrumentation library and RDB run-time engine. For example, the trace-and-replay instrumentation library uses segmentation faults in the region of virtual addresses reserved for machine state storage to map in new anonymous pages of memory to store machine state. These faults generate signals that are not bugs in user code and will be handled internally by the instrumentation library. If the debugger traces the delivery of SIGSEGV, we do not want to have the back-end notify the debugger that the target has stopped on an event of interest if the instrumentation library is handling the signal internally.

Since the debugger only interacts with `/proc` indirectly through the back-end library, the back-end could simply trace all deliveries of SIGSEGV using `/proc`, but continue the process without notifying the debugger if the event is known to be handled by the instrumentation library. Unfortunately this approach violates our goal of limiting the performance overhead of the library because it may cause many needless context switches between the debugger and the target process. The solution to this problem is to modify the run-time framework's signal handling code to also support the notion of traced signals. The run-time engine maintains a set of traced signals, and informs the debugger of the address of the set during startup. The back-end does not use the `/proc` mechanism for modifying the process's traced signal set, but instead modifies the set inside the framework. When a signal is delivered to the process, the framework invokes any instrumentation library callbacks as usual, except that if the library does not handle the signal, and the signal is being traced, the framework breakpoints the process.

The back-end detects that the target has stopped on an event of interest, and queries the framework to determine what type of event has occurred. We can handle other traced events in the framework or instrumentation library in this manner as well. This solves the performance problem because no breakpoint occurs if the signal is handled by the instrumentation library, and thus no context switch between the target and debugger occurs. However, we also need to conceal

the presence of the framework itself from the debugger. At the time the signal is delivered, the routine at the top of the stack is the framework's handler. If the debugger now reports that a segmentation fault has occurred to the user and prints out a stack trace, the user may be confused into thinking the error has occurred inside of the framework. To preserve the transparency of the instrumentation, the framework saves the user context which was originally interrupted by the signal. The back-end can then access this information and report these values back to the debugger when the debugger queries the back-end for the process machine registers.

8 Debugger Front-End and Future Work

To prove that our back-end library can provide the foundation for a real debugging system, we developed a new debugger for Solaris using our back-end. The RDB debugger is a graphical source-level debugger for C language programs which provides an open architecture for the next phase of our work: developing new debugger interfaces and technologies based on the trace-and-replay engine. RDB provides the basic feature set found in real UNIX debuggers: it provides basic process control using breakpoints, it provides display of program source code and correlates the current state of program execution to the source code, it can display a stack backtrace and correlate addresses with program function names, it can display machine register values, and it can display appropriately formatted data using the C types and variable names defined in the program. We designed RDB to be as modular as possible so that adding new displays for viewing or interpreting program and replay information will be easy. The debugger also has an integrated KornShell [11] interface; new debugger features can be implemented or prototyped quickly by adding shell functions. In this section we discuss functionality we have implemented or plan to implement in RDB using our incremental trace-and-replay engine.

8.1 Execution Timeline

RDB supports the control of an arbitrary number of active instances of a particular program being debugged. Each instance (process) can be started in tracing mode or replay mode. The debugger provides a graphical interface for browsing available trace files which can be used to start controlled replay processes, and shows summary information for each trace set including the cause of process termination, the number of intervals, and the machine configuration and date of the original run. We hope to provide a graphical timeline display, initially delineated by interval boundaries, which shows the current location of each active process. The timeline could also be extended to show the location of traced events in each interval, such as system calls or UNIX process signals that were delivered, and would allow the user to manipulate processes on the timeline, causing the selected process to restart replay at the specified interval. If instrumentation to record program control-flow were added to our library, we can envision the timeline evolving into a complete hierarchical control-flow graph of the program with the user able to restart replay from any location on the graph.

8.2 Enhanced Data Display

The RDB data display window provides a graphical display of C data structures in the user program. We have extended this display to allow the user to view the current monitoring machine state

associated with a variable, currently presented as descriptive text, and the associated flowback value and its interpretation. In the future we would like to investigate using color in combination with textual annotations to represent the machine state associated with each variable. We would also like to annotate variables in the system call state with the name and instance of the system call which wrote them, the return value of the call, and allow the user to locate this call in the listing of system calls for the interval or in the timeline view.

8.3 Flowback Queries

The user can select a data item in the data display window and direct the debugger to restart replay at the interval in which the specified value was previously modified. In the future, we would like to enhance this facility using system call trap tracing and `/proc` filesystem data watchpoints to provide improved flowback queries. If we can flow back to the interval where a value was modified, and then set a write watchpoint and continue the program, we can automate the process of rolling execution backward to the precise point where a variable was previously modified, or to the entry of the system call routine which modified the variable. We also hope to provide a graphical interface to allow the user to save a collection of such replay points, or to stop the program using traditional breakpoints during tracing mode and save the current location as a replay point. We envision the programmer creating a collection of important points or bug symptom sites, annotating them with text in the debugger, and then directing the debugger to re-execute the program from these saved locations to observe program behavior.

8.4 Integrated Trace and Replay

We have designed RDB with the intention of transparently supporting trace and replay of a process simultaneously. The basic idea is that the user can execute a program in tracing mode and use standard debugger breakpoints to stop the process. Once the process is stopped, the user can replay the execution of this process from any time up to the stopping point using the trace of the execution thus far. We envision developing instrumentation which can be used to take this idea further and provide the capability for a *backwards step* command in the debugger. We can provide backwards step in the debugger using a coordinated pair of trace and replay processes. When the user issues a backwards step command after stopping the tracing process, we can move backwards a single instruction or series of instructions by using the replay process to flow backward to the start of the previous interval, and then running the replay process forward to the instruction previous to the current location, which we can identify using a (*SIC*, *%pc*) pair.

We could partially provide this capability in our current implementation, but with an unreasonable performance penalty. Consider first the example of backwards step in a basic block, as shown in Fig. 36. The program is stopped at a particular current instruction in a basic block. We can therefore conclude that the previous instruction is located at the address specified by the value of the *%pc* register minus the size of an instruction. To step backwards, we first flow back to the previous interval boundary, and then run the replay forward until we breakpoint at the desired SIC value. We then single-step the process until the *%npc* register addresses the instruction at which we issued backwards step. This provides limited backwards step, but at too great a cost because potentially many context switches must occur between the debugger and the target as we single-step if the basic block is extremely long.

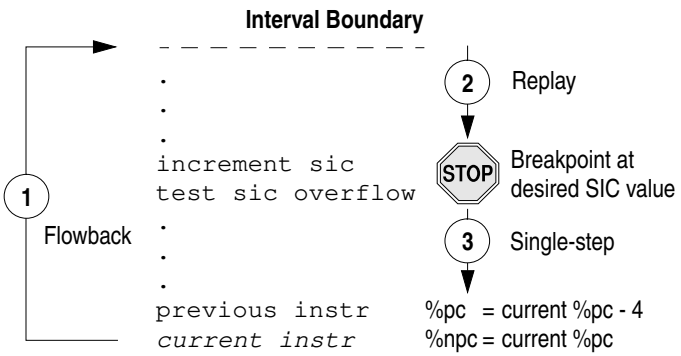

Figure 36: Backwards step in a basic block

We cannot step backwards to a previous basic block because we perform no program *control-flow tracing*. We need to be able to back up more than one machine instruction, including across control-flow transfers, in order to provide complete backwards step at the source level. If the previous source instruction was in another basic block which then branched to the current block, we can no longer reliably compute the address of the previous instruction by subtracting a multiple of the instruction size from the *%pc* register because the branch may have occurred from an arbitrary distance away. We conclude that in order to provide an efficient backwards step capability in the debugger, we must combine our trace and replay techniques with a control-flow trace that incorporates SIC values. We hope to augment our trace-and-replay instrumentation with control-flow tracing to provide another instrumentation option for the programmer that can be used to provide efficient backwards step functionality in RDB.

9 Conclusions

We have presented the details of the design and implementation of RDB, an extensible system for program instrumentation and debugging. We have designed and built a prototype engine for incremental replay and flowback analysis using the RDB architecture. The RDB debugger allows the programmer to robustly instrument and trace the execution of real UNIX programs, including the execution of system calls and interrupts. Using a new incremental tracing technique, we can trace programs and only affect performance by a factor of 2.5–3.5. Moreover, we have sufficiently low trace rates to allow a day-long run to be traced to a gigabyte disk. Our debugger back-end library allows replay to be incorporated into existing debugging tools, and we have built an actual C source-level debugger that incorporates program tracing and replay. The debugger allows the programmer to view program trace information and navigate backward and forward through a program's execution. We have integrated flowback information into the debugger's data display view to allow the programmer to roll back the execution of a program to the point where a data value was previously modified. The debugger provides a powerful tool for tracking program bug symptoms back to their causes. We hope to extend RDB in the future with backwards step capabilities and new interfaces for program debugging and trace visualization.

10 Acknowledgements

The work described in this paper is the result of a two-year collaboration with Professor Robert Netzer. Rob's advice, guidance, and research were essential components of my work on this project. Bryan Cantrill provided inspiration to me throughout our time together at Brown, and reviewed many of the drafts of this document, improving its quality substantially. Daniel Price implemented the stabs processing and data display for the RDB debugger, and contributed the clean parts of its design. Roger Faulkner provided an autographed copy of the Solaris 2.6 `proc(4)` manual page and background information on the agent LWP. Steve Masticola tested many of the RDB components and was instrumental in identifying bugs, contributing to the quality of the system. Jeff Coady and Tom Heft contributed more machines and disks to this project than they will ever know.

References

[1] H. Agrawal. *Towards Automatic Debugging of Computer Programs.* PhD thesis, Purdue University, August 1991.

[2] H. Agrawal, R. DeMillo, and E. Spafford. An execution backtracking approach to program debugging. *IEEE Software*, pages 21–26, May 1991.

[3] J. Choi, B. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[4] R. Faulkner and R. Gomes. The Process File System and process model in UNIX System V. In *Proceedings of the Winter 1991 USENIX Conference*, Dallas, TX, January 1991.

[5] S. Feldman and C. Brown. IGOR: A system for program debugging via reversible execution. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, April 1988.

[6] R. Garner. *The SPARC Architecture Manual: Version 8.* Prentice-Hall, New Jersey, 1992.

[7] J. Gilmore. *Working in GDB.* Cygnus Support, 1.84 edition, 1994.

[8] R. Gingell, M. Lee, X. Dang, and M. Weeks. Shared libraries in SunOS. In *Proceedings of the Summer 1987 USENIX Conference*, Phoenix, AZ, 1987.

[9] R. Gingell, J. Moran, and W. Shannon. Virtual memory architecture in SunOS. In *Proceedings of the Summer 1987 USENIX Conference*, Phoenix, AZ, 1987.

[10] D. Hanson and M. Raghavachari. A machine-independent debugger. Technical Report CS-TR-500-95, Princeton University, November 1995.

[11] D. Korn. ksh: An extensible high level language. In *Proceedings of the Very High Level Language Symposium (VHLL)*, Santa Fe, NM, October 1994.

[12] J. Mellor-Crummey and T. LeBlanc. A software instruction counter. In *Proceedings of the Third ASPLOS*, April 1989.

[13] R. Netzer. Personal communication, October 1995.

[14] R. Netzer. Personal communication, May 1996.

[15] R. Netzer and M. Shapiro. Efficient fine-grain monitoring and its application to trace-and-replay debugging. Unpublished extended abstract, November 1996.

[16] R. Netzer and M. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[17] J. Plank, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Submitted for publication, 1995, May 1996.

[18] M. Powell, S. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter 1991 USENIX Conference*, Dallas, TX, January 1991.

[19] M. Shapiro. Tracing of user programs for incremental replay. Unpublished senior thesis, April 1996.

[20] R. Stallman. *Debugging with GDB.* Cygnus Support, 4.12 edition, January 1994.

[21] Sun Microsystems, Inc., Mountain View, CA. *Linker and Libraries*, 1995.

[22] U. Vahalia. *UNIX Internals: The New Frontiers.* Prentice-Hall, New Jersey, 1996.

[23] L. Wall and R. Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, CA, 1991.

[24] D. Weaver and T. Germond. *The SPARC Architecture Manual: Version 9.* Prentice-Hall, New Jersey, 1994.