

Tracing of User Programs for Incremental Replay *

Michael W. Shapiro

Dept. of Computer Science

Brown University

mws@cs.brown.edu

April 20, 1996

Abstract

User programs may be long-running and have arbitrarily complex interactions with their environment, and thus reproducing bugs by repeated re-execution may be impractical or impossible. To facilitate the debugging process, we hope to produce a tool which allows the user to efficiently trace the execution of a given program, and then exactly replay any interval of the traced execution. In this paper we show how the technique of tracing unique-spanning reads can be further refined to efficiently trace system calls during program execution using a new state machine to detect and trace reads following system calls. We also discuss how interrupts can be traced, and how the trace information can be used to incrementally replay a program execution, including system calls executed by the program and interrupts which occurred during the original execution.

1 Introduction

Debugging requires repeated executions of a given program. 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. Programmers need be able to reproduce an exact execution by *tracing* a given execution of a program, and then exactly *replaying* this execution from the traced information. In addition, the tracing and replay process must not be prohibitively expensive in terms of the cost to execution time, the cost to replay, and the space required to store trace information. Finally, because a given execution may run for a long time before a given bug is exercised, execution replay tools must allow the programmer to replay a program *incrementally* from frequent intermediate checkpoints.

In order to reduce trace size and the run-time overhead required by tracing, Netzer and Weaver [5] identified a class of *unique-spanning* memory accesses. 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* trace records produced prior to the

*Senior Honors Thesis for Computer Science 193, 194: Prof. Robert H.B. Netzer. Copyright ©1996 Robert H.B. Netzer and Michael W. Shapiro. Brown University Confidential—Do not copy or distribute.

current interval. Their work demonstrated that dividing a program's execution into fixed-size intervals and adaptively tracing the unique-spanning reads within each interval can be done efficiently, and provides sufficient trace information to allow incremental replay. Experimental results [4] have shown that this technique can be implemented with about a factor of two slowdown and produce only 10–20 kilobytes of trace per second.

In this paper we show how this technique can be further refined to efficiently trace system calls during program execution using a new state machine to detect and trace unique-spanning reads following system calls. We also discuss how interrupts can be traced efficiently along with memory accesses, and how the trace information can be used to incrementally replay a program execution, including system calls executed by the program and interrupts which occurred during the original execution. Finally, we discuss the types of debugging tools we hope to construct from an incremental trace-and-replay engine in the future.

2 Tracing

In order to provide incremental replay, we divide a program's execution into fixed-size intervals of a predefined length T . During each interval, we need to trace sufficient information so that any given interval can be replayed by scanning at most the M previous interval trace records. Each interval's replay is therefore said to be *M-bounded*. Providing bounded-time replay of any interval is a necessary condition for any practical incremental replay tool. An execution which crashes after a day long execution could not be effectively debugged if a debugging query about the state of the program just prior to the crash required replay of a day's trace data.

2.1 What to Trace

We can correctly replay any interval if each read from memory receives the same value as during the original execution [5]. Netzer and Weaver [5] concluded that for *M-bounded* replay, we only need to trace *unique-spanning_M* reads. A *unique-spanning_M* read is the first read from an address in a given interval where this address's value was not traced in the previous M intervals. In the next section we show the development of new finite state machines which determine whether or not a given address should be traced at the end of each interval. These state machines adaptively determine whether each read is unique-spanning.

Netzer and Weaver have also explored the possibilities of useful values of M , and whether tracing unique-spanning *writes* could also be used to support incremental replay [5]. Although tracing unique-spanning writes requires less run-time overhead, it requires the ability to restore state from *all* previous intervals in order to effect replay. This work is therefore focused on efficient tracing of unique-spanning reads in order to produce a useful and efficient mechanism for both tracing and replay. In addition, the state machines we present trace only *unique-spanning₀* reads. These machines are the most straightforward to design, and produce trace data which requires the least setup time to replay. However, all of the designs presented can be extended to support tracing for other values of M .

2.2 When to Trace

Ideally, the amount of trace data can be minimized by placing interval boundaries so that the number of unique-spanning_M reads is minimized. Previous work [4] has shown that this can be performed in $O(nT)$ time, where n is the number of memory references. However, to avoid performing this computation during program execution and simplify the tracing implementation, the length of each interval can simply be fixed at T seconds, a parameter specified by the user. This can be implemented efficiently by using the interval timer facility provided by many operating systems to deliver an interrupt to the process after the specified time quantum has elapsed.

2.3 Detecting Unique-Spanning Reads

In order to facilitate efficient, adaptive tracing of unique-spanning reads, Netzer developed a technique where a finite state machine is associated with each memory address [4]. Each read or write to this address causes a transition to be taken in the machine. An additional transition is taken in each machine at the end of an interval. A state transition may include an action to trace the value associated with the address. The state machine is designed to only trace unique-spanning reads, and thus ensures that no unnecessary information is traced. Netzer's [4] two-state DFA for tracing unique-spanning reads is shown in Fig. 1.

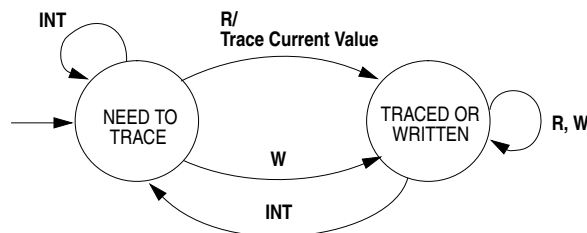


Figure 1: Two-state machine for tracing unique-spanning₀ reads

Before the program begins executing, and at each interval boundary, the state machine associated with each address is initialized to the *Need to Trace* state. During an interval, if a read occurs, a read transition (denoted R in the figure) is taken in the state machine associated with this address. If this machine is in the *Need to Trace* state, this is the first read of the address in this interval; we trace the current value stored at this address and transition to the *Traced or Written* state. If we have already traced this address and another read occurs, the read transition causes the machine to remain in the same state and no tracing occurs. However, if the value is first written (denoted W in the figure) in the interval, then we can immediately transition to the *Traced or Written* state. We do not need to perform any additional work because this write will produce the value returned to subsequent reads when the interval is re-executed.

Although this state machine correctly traces only unique-spanning₀ reads within each interval, this design has a major performance problem: each read from memory requires a check to determine the current state, and optionally requires the execution of code to trace the current value stored at the given address. Previous experimental results demonstrate that reads occur two to ten times more frequently than writes, and that both reads and writes occur millions of times more frequently than interval boundaries using fixed interval lengths of a few seconds [5].

In order to minimize the runtime overhead of tracing, we must develop state machines which

reduce the amount of work done for the most common transitions. We will show a series of state machines which offer improvements over this basic design, leading up to the design of the machine using for our prototype implementation. We first examine how states can be represented during program execution, and what other events must be traced during execution in order to effect replay.

2.4 Representing States

In order to maintain the state of each address during program execution, we associate a unique value with each state. The value indicating the current state of the machine for a given address is stored in a fixed-size bit vector associated with each address. To determine the size of the bit vector we consider the number of bits needed to uniquely identify each state of our machine, and the ease of locating and extracting the needed bits for a machine given a memory address. Although a given DFA may only require a small number of bits to uniquely identify each state, we may choose to reserve up to an entire word for each DFA state value in order to improve the performance of our hooks. If multiple DFA state values are packed together in the same byte or same word, our hooks will require additional instructions to extract the bits for a single DFA state from a byte, word, or half-word loaded into a register. This will require less overall storage, but will increase the run-time overhead of the tracing hooks. We will discuss what other useful information might be associated with a given address in order to enhance replay capabilities if more bits are associated with an address than are necessary to store the DFA state value.

Two methods of representing DFA bit vectors have been considered: two-level bit vectors and flat bit vectors [4]. We associate a state machine with each word of memory in the address space because most memory accesses are word accesses, and because 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 [1]. Tracing at a lower level of granularity, such as bytes, could also be done, but our goal is optimize for the common case of full-word memory accesses.

Fig. 2 shows an example two-level bit vector implementation. The topmost bits of an address are used to index into a table of pointers to chunks of bit vectors. Each chunk contains a sequence of bit vectors storing the state associated with each address. Using the example sizes in Fig. 2 and assuming a 32-bit address space, we require 256 4-byte pointers for our top-level table, consuming 1K of required space. Each chunk must store the state for 2^{24} machines, and each machine might require one byte of state per machine, so each chunk requires 16K. Overall, this method would require 4MB of storage for DFA chunks if the entire table were allocated. Since DFA chunks can be dynamically allocated, this method is very space efficient even when an entire byte is used to represent each DFA state, as shown in Fig. 2. The chunks also exhibit the same locality of reference as the addresses referenced by the user program itself. If a larger unit such as a word is required to store each DFA state, we can achieve better run-time performance by using a *flat* bit vector and avoiding the need to dereference a pointer for each DFA access.

Netzer [4] has shown that a flat bit vector can also be used to represent DFA states, as shown in Fig. 3. In this approach, we simply associate a word of DFA state with each word addressed by the user program. This approach performs better than the two-level approach because we can more quickly determine the address of the state word from a given address which is written or read. However, the flat bit vector approach requires significantly more storage space because in the worst case, half the address space will be touched by the user program, and the other half will be allocated to store bit vectors. This requirement may not be unreasonable when we consider that

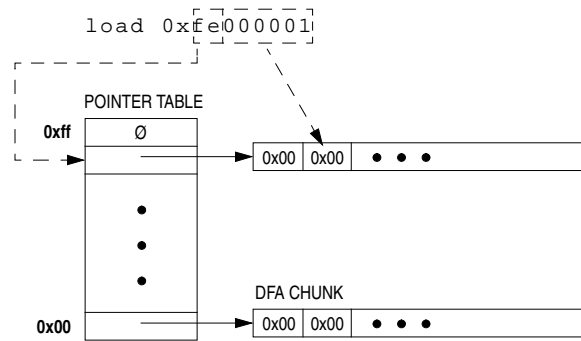


Figure 2: Two-level bit vector using one byte to encode the state for each accessed word

a 32-bit address space is 4GB in size; a program traced using the flat bit vector approach still has 2GB of address space available. We can also sparsely allocate the flat bit vector by detecting page faults in the region of memory occupied by the bit vector, and mapping in new pages of initialized bit vectors as needed.

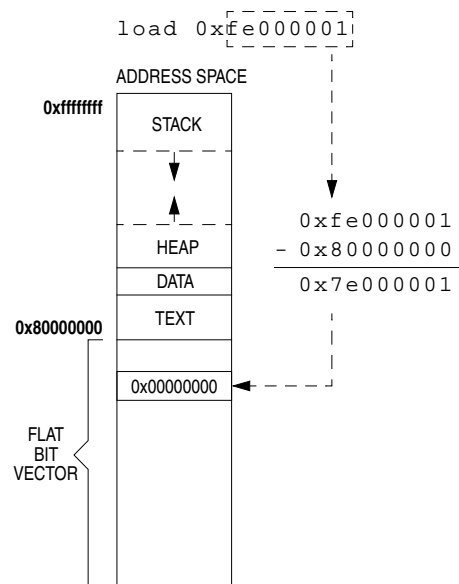


Figure 3: Flat bit vector using one word to encode the state for each accessed word

3 Other Tracing Issues

The SPARC-based prototype developed for this research includes an assembly-language instrumentation tool which is used to scan the assembly output of the compiler before it is passed to the assembler and add instrumentation code to instructions of interest. In this section, we consider some specialized tracing issues which affect our instrumentation for the SPARC. We develop techniques to solve these problems which are applicable to many other operating systems and architectures.

3.1 The Stack

Register-based load-store architectures such as the SPARC have special instructions for saving and restoring registers to the stack. Whenever registers are restored from the stack during a program's execution, we must consider these restores to be reads from memory which may need to be traced. This issue is complicated on the SPARC by the use of overlapping register windows.

During the execution of a program, a given SPARC instruction may access 8 global registers and a *window* of 24 registers in the set of general-purpose registers [2]. If all of the available register windows are in use and a **save** instruction is executed, a window overflow trap occurs and register windows are saved to the stack. Similarly, a **restore** when no register windows are in use causes a window underflow trap which restores register values from the stack. These instructions are therefore difficult to analyze since they may or may not cause memory accesses.

To guarantee correct replay, we must be sure that any **restore** instruction which reads register values from the stack receives the values written there by the corresponding **save**. There are two cases of interest: a **save** and **restore** pair in the same interval, and a **save** and **restore** pair which crosses an interval boundary. In the first case, we know that correct replay of the interval guarantees that the same values will be present in the registers which are written to the stack when the **save** is issued in the replay and the original execution. If the **save** causes window overflow, then the **restore** will cause window underflow and read the values which were written, and thus this case can be handled without any additional instrumentation.

We must trace some additional information in order to provide correct replay for a **save** and **restore** pair which is divided by an interval boundary. If register values written by the **save** instruction are not traced, then we cannot guarantee correct incremental replay for the interval which contains the **restore** because the register values read by the **restore** will not have been restored into the user address space. To avoid the overhead of specialized instrumentation of the **restore** instruction, we instead trace the entire user stack at each interval boundary. During program startup, we record the address of the bottom of the user stack (the topmost stack address for the SPARC). At each interval boundary, we execute a trap into the kernel directing the operating system to flush all of the register windows to the stack, and then we write the entire range of addresses between the bottom of the stack and the current value of the stack pointer to the trace file. This does not add too much burden to the tracing: A single **write** system call writes the trace because it is a consecutive range of addresses, and the user stack is typically small for most programs, except for some specialized applications such as recursive-descent parsing.

This technique of tracing the stack guarantees correct replay for all cases except for user programs which contain code to read from the uninitialized area of the stack reserved for register window dumps. Although such code indicates the presence of programmer error, we cannot guarantee correct replay of the values returned from such reads. Traps into the kernel may occur at any time, potentially writing register values to the stack in the case of window overflow. We cannot determine if window overflow will occur because these traps cannot be predicted, and so reads from this area may return either the uninitialized values on the stack, or the register values written there by a window overflow trap. We may offer the user the option of additional instrumentation code to detect reads from these areas of the stack in order to locate these bugs.

3.2 System Calls

User programs execute system calls by trapping into the kernel. The code executed by the kernel on behalf of the user process may write to memory in the user address space, and our tracing engine must be able to trace subsequent reads from the affected addresses so that these reads can be replayed. To simplify tracing, we can instrument each trap so that a system call transition is taken in the state machine associated with each word of memory which could potentially be affected by the system call. Determining the range of affected addresses is straightforward: for example, the arguments to a UNIX `read` system call specify the starting address of a buffer where data is to be written and the length of this buffer.

System call transitions for the state machine of Fig. 1 are added in Fig. 4. By changing the state of each machine whose address was affected by the call to the *Need to Trace* state, we guarantee that subsequent reads of addresses affected by the system call will be traced. Another possibility would be to simply trace all addresses potentially affected by the system call immediately after the call, but this has the disadvantage of increased tracing overhead and increased trace size. We will examine several different methods of handling system call transitions in our DFA designs, and consider how each will affect tracing performance and replay.

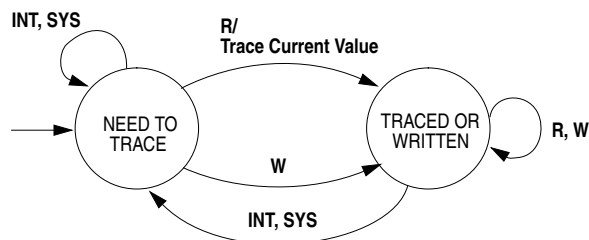


Figure 4: Two-state machine with system call transitions

3.3 Interrupts

Software interrupts, such as those generated by UNIX process signals, must also be traced during program execution so that they can be replayed properly. We can add a software instruction counter (SIC) to the user program as part of our instrumentation so that the unique $(SIC, ProgramCounter)$ pair can be traced whenever an interrupt occurs. Mellor-Crummey and LeBlanc [3] have shown that a SIC can be implemented by adding instrumentation code to increment a counter on backwards branches and procedure calls with less than a 10% run-time overhead on average. Their implementation for the SPARC used two instructions: one to decrement a register and another to optionally trap if register underflow occurs during replay. We improve on this idea by only using a single instruction to increment a register during program execution. Later we discuss how we can use code patching prior to the re-execution of each interval to correctly replay the interrupts. Maintaining a SIC is also useful for determining where our interval boundaries occurred. The interval timer used for our prototype generates a software interrupt to indicate that the timer has expired, and so we can trace the SIC and PC values for the end of each interval as part of the same mechanism.

4 State Machine Design

The state machine presented in Fig. 1 is not practical for implementing a run-time tracing engine because of the overhead required for the read hook. Netzer [4] designed an improved machine which requires four states, and postpones all tracing until interval boundaries. This machine is shown in Fig. 5. By postponing tracing until interval boundaries, this machine requires much less run-time overhead than the original two-state DFA. Netzer also devised a 3-bit numbering assignment for the states of this new machine which allows the read and write hooks to perform identical, simple bit manipulations in each state [4]. This allows for straightforward and efficient instrumentation of the user program. At the assembly level, the instrumentation tool can insert instructions to perform a simple bit manipulation to the state information associated with a given address after each load or store instruction. The instrumentation code contains no conditional tests or branches because it is not necessary to perform different actions for different input states.

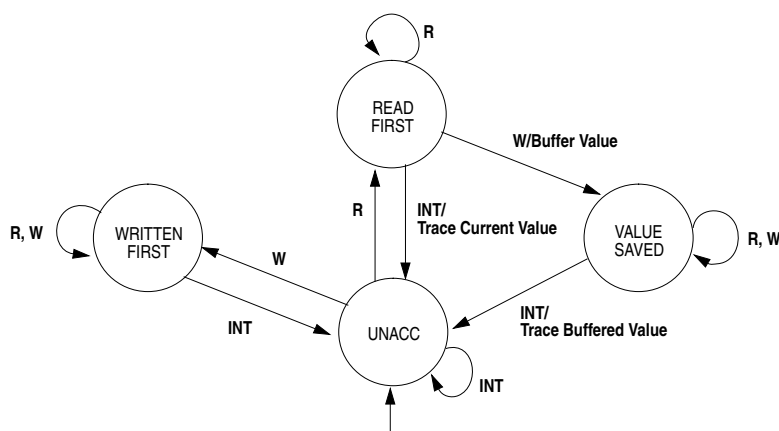


Figure 5: Four-state DFA for tracing unique-spanning₀ reads. All tracing is performed at interval boundaries.

We now examine how this four state machine can be extended to handle system calls while still keeping the run-time overhead as low as possible. We compare the advantages and disadvantages of several approaches, and then develop a final design into a complete implementation for the SPARC.

4.1 Trace System Calls

One possible approach for handling system calls is to simply trace all of the potentially affected addresses at the site of the system call. Fig. 6 shows the four-state machine from Fig. 5 with system call transitions added. This DFA retains the advantages of the original four-state machine for reads and writes, but has some potential performance problems. First, we generate unnecessary trace data by immediately tracing all potentially affected addresses. Second, our system call hooks become more expensive because they may have to additionally trace two values for each address if the state machine is in the *Read First* or *Value Saved* states. In each of these states, we have already determined that a unique-spanning read occurred for this address and that it should be traced, but we are trying to postpone the tracing until the next interval boundary. If a system call subsequently overwrites this value, we need to first trace the current value, then execute the system call, and then trace the new value.

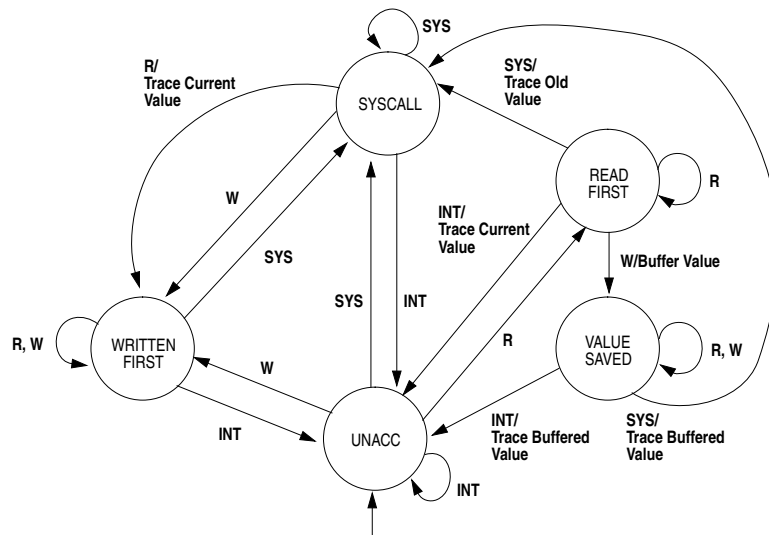


Figure 7: DFA for tracing unique-spanning₀ reads with an additional system call state

to postpone the trace of the read until the next interval boundary. The common case is fast because the read hook need only perform a state transition. Our trace size is optimal because we only trace addresses which are read following system calls. The system call transition needs to optionally emit a trace record in order to preserve a value which may be overwritten by a system call, but the added work is outweighed by the cost of trapping into the kernel for the system call itself.

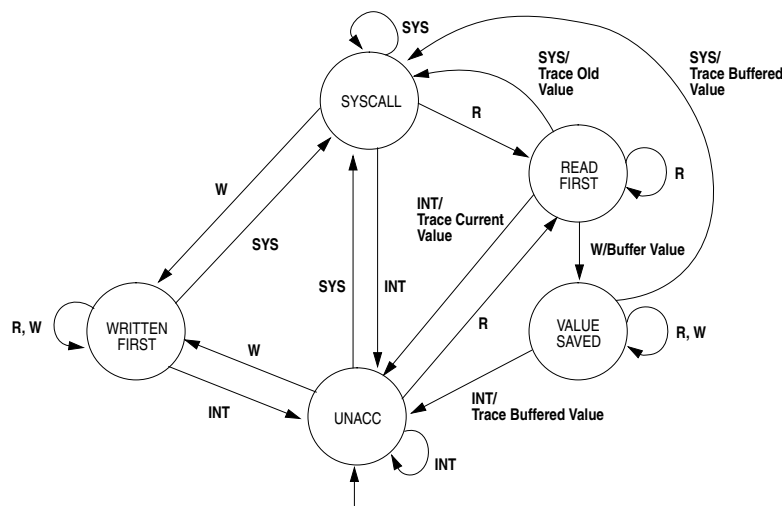


Figure 8: DFA with system call state which traces on some system call transitions

This design still suffers from the problem of out-of-order trace records. In particular, we have no way of knowing whether values traced when an interval boundary transition occurs in machines in the *Read First* and *Value Saved* states should be restored at the beginning of the interval during replay, or should be restored following a system call. To solve this problem, we need to be able to identify which trace records are associated with system calls at run-time.

4.4 Identifying System Call Traces

We can extend the design of Fig. 8 to include two additional states to detect values which are traced following system calls. The *Read After Syscall* and *Saved After Syscall* states shown in Fig. 9 mirror the use of the *Read First* and *Value Saved* states, but these states are only used once a transition has been taken to the *System Call* state. This design maintains all the benefits of the previous five-state machine, but now we can examine the state of the machine when our system call and interval boundary transitions occur to determine if a trace record should be restored for replay at the beginning of the interval or after a system call. Trace actions marked in the figure with an asterisk (*) indicate traces following system calls.

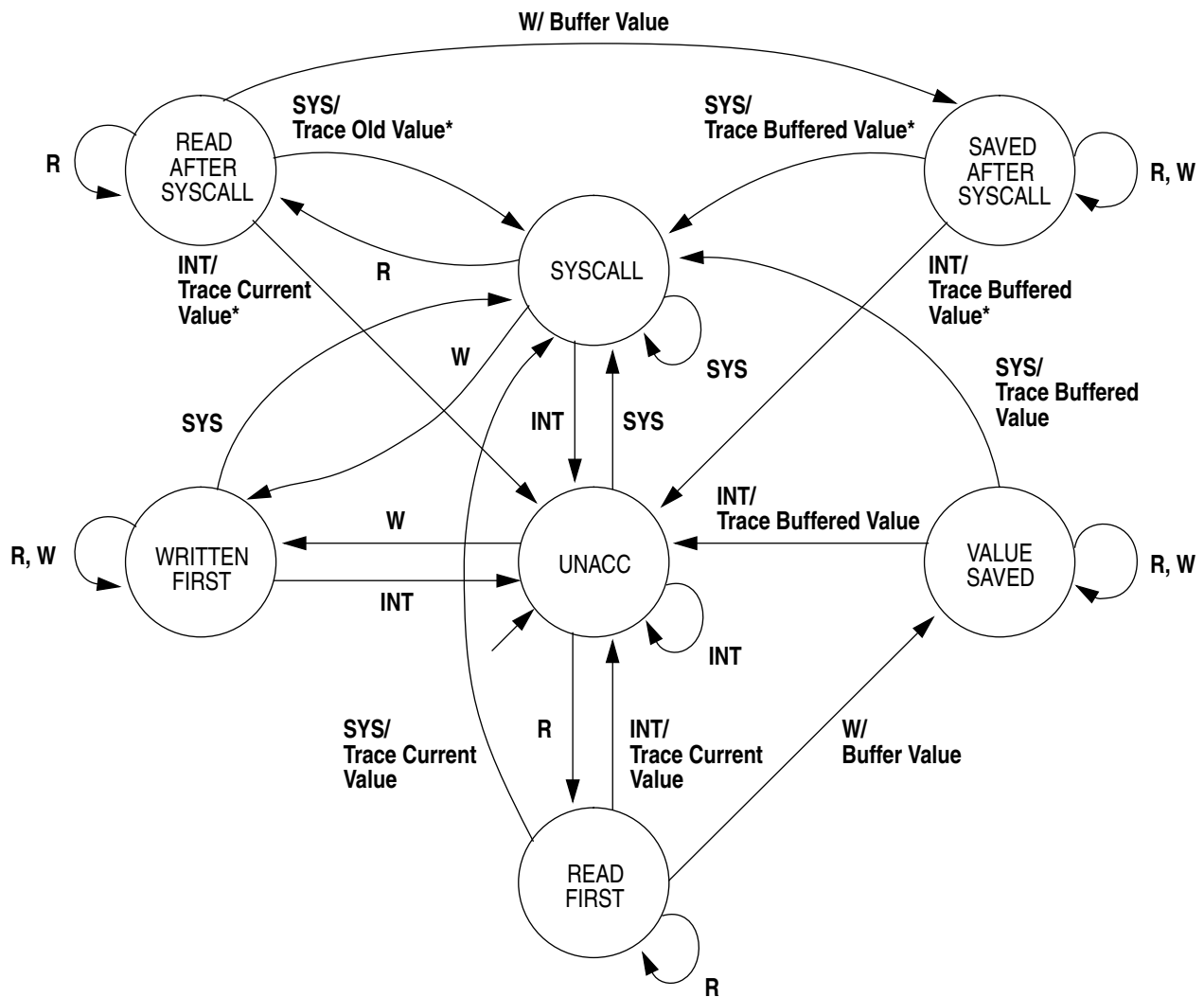


Figure 9: DFA which identifies traces which are restored at the beginning of an interval and traces due to system calls (indicated with trailing “*”)

The trace records produced as a result of system calls may be tagged in the trace file, or we can produce two separate trace files. In either case, the tagged trace records are ordered with respect to the order of the system calls in both execution and replay. We now turn our attention to adding additional transitions needed for our prototype on the SPARC, and devising a bit assignment for

the states so that our transition hooks will be as fast as possible.

5 State Machine Implementation

To implement our prototype, we must extend our DFA design to handle instructions available on the SPARC which allow partial-word memory accesses and register-to-memory swaps. We also need to determine a mapping of values to DFA states which will produce fast instrumentation code.

5.1 Partial Word Transitions

We have already shown two approaches for storing DFA states in which one state value is associated with each word of memory in the user program. Our machine includes transitions for each read or write of the associated word of memory. However, the SPARC assembly language and the assembly for many other architectures includes instructions to access memory at the byte or half word granularity. Our DFA must therefore include transitions to handle the case of a word which is only partially read or written. The DFA from Fig. 9 with transitions for partial-word reads (indicated by PR) and partial-word writes (indicated by PW) is shown in Fig. 10.

In general we consider partial-word reads to be equivalent to full-word reads. Since we must be able to replay all reads correctly, no matter which part of the word is read, we must trace the entire word in order to properly restore its value. We do not change states when a partial write occurs because a portion of the word which is not overwritten may subsequently be read and replaying the partial write will not restore the value; it must be traced. We only take action on a partial write when the current value has been identified as the value returned by a unique-spanning read. In this case, the partial write will overwrite some portion of the value we need to subsequently trace, and so we must buffer the current value before proceeding with the partial write.

System calls may only write a portion of a word in the user address space as well. Our DFA traces reads of addresses which have been affected by system calls, so we can handle a partial-word system call transition exactly as a full-word system call and be able to replay correctly. This may result in some unnecessary trace in the case of a given byte within a word being written by a system call and a different byte of the same word being subsequently read, but this overhead should be relatively small, and our approach avoids adding any extra complexity to the DFA.

5.2 Swap Transitions

The SPARC instruction set provides two instructions for performing swap operations: `swap` and `ldstub` [2]. The `swap` instruction is a full-word register-to-memory swap which exchanges a word between a specified register and specified address. The `ldstub` instruction atomically loads the specified register with the lower-order byte of a given address and writes the value 255 to that byte. For the purposes of our state machine, we can consider the first to be equivalent to a full-word read followed by a full-word write, and the second to be equivalent to a partial-word read followed by a partial-word write.

We can therefore instrument each instruction with the hook transitions and actions associated with a read and a write, or a partial-word read and a partial-word write. We also consider that

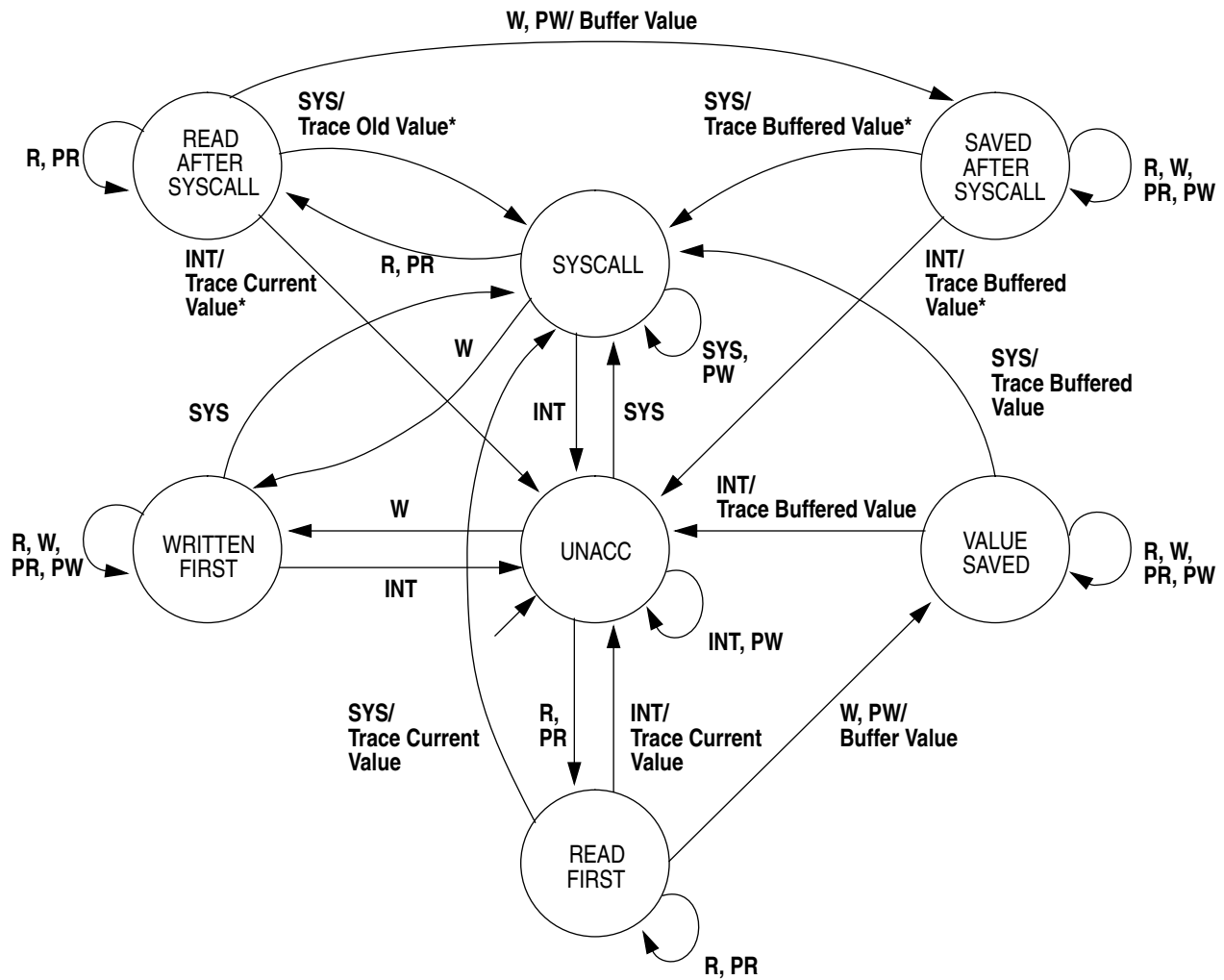


Figure 10: DFA with partial-word transitions added

an additional transition could be added to each state for swaps. Examination of our DFA shows that for all states, performing a full-word read followed by a full-word write always results in the same destination state and associated actions as a partial-word read followed by a partial-word write, so the two can be considered equivalent for the purposes of our DFA. In the next section, we demonstrate that our bit layout offers an optimized transition for the swap instructions.

5.3 DFA Bit Layout

In order for our assembly-language hooks to operate efficiently, we want to avoid any type of conditional tests to determine the destination state for a given transition. To this end, we develop an assignment of bit patterns to the states of our DFA which allows a sequence of simple arithmetic manipulations to be performed for each type of instruction to be instrumented, regardless of the current state. Fig. 11 summarizes our assignment of bit patterns to states. Bits marked with an asterisk (*) indicate that the value of that bit may be either a zero or a one.

Fig. 12 shows the complete pseudo-code for our tracing engine instrumentation using the spec-

<i>State</i>	<i>Bit Sequence (ABCDE)</i>
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 11: Bit assignments for DFA states

ified bit layout. The instrumentation adheres to our philosophy of making the common case fast, and only performing complex work at system calls and interval boundaries. The instrumentation code does not include any explicit code to buffer values, except in the system call transition. In the next section, we show how all buffering performed following write and partial-write transitions can be handled implicitly using virtual memory support provided by the operating system.

We can make some observations about the design decisions involved in our bit layout based on the requirements imposed by the state machine. We must be able to distinguish between the sequence read-write (RW) and write-read (WR): the first sequence contains a unique-spanning read and must be traced, the second does not. If the read and write hooks operated only on separate state bits, we would be unable to distinguish this sequence; therefore the read and write hooks must operate on some common bits. If both the read and write hooks consisted only of bit sets or bit resets, there would be way no way to distinguish the sequence; therefore one of the read or write hooks must set one or more bits and the other must reset one or more bits.

We must also be able to distinguish the sequence WR from RWR; the first requires no tracing, the second contains an initial unique-spanning read which must be traced. If the read and write hooks consist only of sets and resets, we cannot distinguish these two sequences since the result of the initial read action will be undone by the subsequent write and read actions in the RWR sequence. Therefore the read and write hooks must contain some other logical operation aside from set and reset.

Our DFA design requires that for some states, a sequence of read transitions or write transitions be idempotent. For example, a write in the *Unaccessed* state causes a state transition to the *Written First* state. Any number of subsequent writes causes no state change once we have entered the *Written First* state. The logical *or* fits this requirement perfectly. If we logically *or* a zero bit with a one, we obtain a one. If we subsequently *or* this one with a one, the result is still one. A proposed subject of future work is to more precisely define the mathematical rules governing bit layouts and transition actions in order to determine the optimal bit layout for a given DFA.

6 SPARC Prototype

In this section we show the additional design considerations involved in building a real prototype of our DFA design for the SPARC architecture. We show how virtual memory copy-on-write techniques can be used to provide buffering, how the DFA bit layout can be implemented to provide the smallest possible instruction sequences, and we present sample instrumentation code.

```

interval_boundary_hook {
  foreach address {
    if((AB) == 00)
      write_trace(state, address);
    (ABCDE) = 01110
  }
}

system_call_hook {
  foreach address potentially written {
    buffer_value(address);
    if((AB) == 00)
      write_trace(state, address);
    (ABCDE) = 01101
  }
}

write_trace(state, address) {
  if((C) == 0)
    value = buffered_value(address);
  else
    value = current_value(address);
  if((E) == 0)
    trace_normal(address, value);
  else
    trace_syscall(address, value);
}

read_hook {
  (BD) = 00
}

write_hook {
  (A) |= (B)
  (BCD) = 001
}

partial_read_hook {
  (BD) = 00
}

partial_write_hook {
  (CD) = 01
  (C) |= (B)
}

swap_hook {
  (BC) = 00
}

initialization_hook {
  foreach address {
    (ABCDE) = 01110
  }
}

```

Figure 12: Pseudo-code for instrumentation hooks

6.1 Buffering

We wish to avoid a conditional test in the full and partial-word write hooks to determine if we should buffer the current value stored in a given address. We take advantage of the copy-on-write virtual memory facility provided by the operating system to perform this buffering automatically. At each interval boundary, we execute the `fork` system call to create a copy of the user process. The `fork` provided by the Solaris operating system used for our prototypes does not actually copy all of the parent process's virtual memory pages at the time of the `fork`; the pages are marked as copy-on-write, and the virtual memory system defers copying any page until the parent or child faults on a given page [7]. Therefore we effectively instruct the operating system to automatically buffer *any* value in the child's address space the first time it is modified by the parent during the interval.

At the end of each interval, we scan each address in the region of allocated DFA bit vector pages and fetch the buffered value from the child process for all machines which are in the *Value Saved* and *Saved After Syscall* states. Once all traces for this interval have been written, the child process is terminated and a new child is forked prior to the execution of the next interval. Any DFA in the *Value Saved* state corresponds to an address which was read first during an interval, and subsequently overwritten; the original value was buffered when the virtual memory system copied the page of memory from the parent process to the child when the parent attempted to write to the address. However, because system calls may occur at any time during the interval, we may repeatedly cycle through the *System Call*, *Read After Syscall*, and *Saved After Syscall* states. Since the virtual memory technique only offers buffers the first overwritten value in the interval, we must perform additional work to buffer values for the transition between *Read After Syscall* and *Saved After Syscall*.

A system call itself is an expensive operation, and we already must perform optional tracing work in the system call transition. Therefore, we would prefer to perform any needed buffering at the site of the system call, so as not to increase the overhead of the full and partial-word write hooks. In order to perform the system call transition in the DFAs for all addresses potentially affected by the system call, we must be able to compute all addresses which may be overwritten by the system call. Once this is done, we can simply write all of the values stored at these addresses to the corresponding addresses of the controlled child process, effectively buffering them. If these values are subsequently overwritten in the parent, our routine to obtain a buffered value from the child process given an address will obtain the value written by the system call.

6.2 Bit Layout

To minimize the overhead of the instrumentation code, we implemented the flat bit vector scheme for our SPARC prototype. The lower half of the address space is reserved for our bit vectors when the user program is linked using a link-editor map provided as input to the Solaris link-editor [6]. Our DFA only requires five bits to track the current state, but our flat bit vector allocates one word (32 bits for our SPARC prototype) of DFA state for each address. We propose storing an interval number or other trace state in the extra bits for future debugging capabilities. Our layout of bits within the bit vector word is shown in Fig. 13.

We consider two factors in determining the bit layout: First, we want the read hook to be as fast as possible. Second, the SPARC arithmetic instructions are limited to 13-bit immediate values [2],

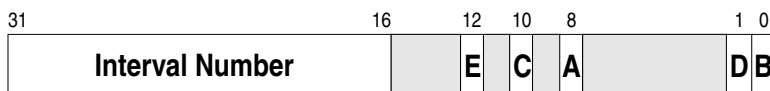


Figure 13: Layout of DFA bits within each bit vector word

so we want to locate all of the bits within the low 13 bits of the word so that any bit masks we need can be used as immediate values and do not need to be pre-loaded into registers. The bit layout in Fig. 13 achieves both of these goals: All bits for the DFA are located in the bottom 13 bits of the word, and the **B** and **D** bits used by the full and partial-word read hooks are isolated in the low-order byte of the DFA word. As shown in the next section, this allows these hooks to clear both bits with a single SPARC instruction.

6.3 Instrumentation Code

We now examine the actual instrumentation code used for the prototype tracing engine for the SPARC. To further optimize our hooks, we take advantage of the global registers available to the user program, denoted `%g1` through `%g7` in the assembly language. We use these registers as scratch space in our hooks to avoid having to save and restore the values in registers being used by the user program. Fig. 14 summarizes our use of global registers. We also avoid any instructions we might modify the condition codes stored in the Processor Status Register so that these do not have to be saved and restored.

Register Name	Use
<code>%g1</code>	Unused by hooks
<code>%g2</code>	Temporary for state computations
<code>%g3</code>	Temporary for state computations
<code>%g4</code>	Temporary for address computations
<code>%g5</code>	Store constant value 0x80000000
<code>%g6</code>	Store constant value 0x80000003
<code>%g7</code>	Software Instruction Counter

Figure 14: Global register usage for SPARC prototype

The SPARC assembly language allows the programmer or compiler to specify addresses for loads and stores by using a single register, a signed 13-bit immediate value, the sum of a register and signed 13-bit immediate value, or the sum of two registers [2]. We present the original instruction and added instrumentation for example instructions in Fig. 15. These examples are for the simplest case: an address specified by a single register. Addresses specified by the sum of two registers or a register and a signed immediate require an additional instruction to add the two address operands into a temporary register.

7 Replay

We now discuss how to perform incremental replay of the user program from our trace data. It is important to note that we use the *same* instrumented code for replay as we used for execution;

<i>Original Instruction</i>	<i>Instrumented Code</i>	<i>Comments</i>
ld [%rY], %rX	stb %g0, [%rY+%g6] ld [%rY], %rX	! (BD) = 00 ! Full-word read
ldub [%rY], %rX	andn %rY, 0x3, %g4 stb %g0, [%g4+%g5] ldub [%rY], %rX	! Word-align address ! (BD) = 00 ! Partial-word read
st %rX, [%rY]	st %rX, [%rY] ld [%rY+%g5], %g3 and %g3, 0x1, %g2 sll %g2, 8, %g2 or %g3, %g2, %g3 or %g3, 0x403, %g3 xor %g3, 0x401, %g3 st %g3, [%rY+%g5]	! Full-word write ! Load DFA state ! Isolate (B) ! %g2 <<= 8 ! (A) = (B) ! (BCD) = 111 ! (BC) = 00 ! Store DFA state
stb %rX, [%rY]	andn %rY, 0x3, %g4 stb %rX, [%rY] ld [%g4+%g5], %g3 or %g3, 0x402, %g3 xor %g3, 0x400, %g3 and %g3, 0x1, %g2 sll %g2, 10, %g2 or %g3, %g2, %g3 st %g3, [%g4+%g5]	! Word-align address ! Partial-word write ! Load DFA state ! (CD) = 11 ! (C) = 0 ! Isolate (B) ! %g2 <<= 10 ! (C) = (B) ! Store DFA state
swap [%rY], %rX	ld [%rY+%g5], %g3 andn %g3, 0x401, %g3 st %g3, [%rY+%g5] swap [%rY], %rX	! Load DFA state ! (BC) = 00 ! Store DFA state ! Full-word swap
ldstub [%rY], %rX	andn %rY, 0x3, %g4 ld [%g4+%g5], %g3 andn %g3, 0x401, %g3 st %g3, [%rY+%g5] ldstub [%rY], %rX	! Word-align address ! Load DFA state ! (BC) = 00 ! Store DFA state ! Load-store unsigned byte

Figure 15: Sample DFA hooks for SPARC prototype

otherwise function pointers may not have the proper value during replay as they did during execution [5]. This also means that our tracing hooks to modify the DFA state are also running during replay; this is acceptable because of the low overhead of our tracing technique. In addition, any additional information produced by the tracing engine in addition to updating DFA states will be automatically reproduced without it having to be traced to or restored from disk. In particular, if our tracing engine stores the interval number of the last interval in which an address was modified in the high-order half word of the bit vector word, this information will be available to our replay tool. We hope that such information can be used in the development of a tool to handle flowback debugging queries.

7.1 Restoring Trace Data

In order to provide incremental replay of the user program from our trace data, we must restore the values from the trace file for each interval to the user address space. The standard trace file contains a trace of the user stack prior to the execution of the interval, a sequence of *(address, value)* pairs to restore to the user address space, and a record indicating the end of each interval. Prior to the replay of each interval, we must restore the user stack, reset the stack pointer, and restore each of the saved values to the appropriate addresses.

We must perform additional work during the replay of a given interval in order to correctly reproduce the system call results of the original execution. Our system call trace file contains a *(SIC, PC)* pair for each system call, followed by a sequence of *(address, value)* pairs recording the values potentially written as a result of the system call in the original execution. At each system call, our replay code checks the current value of the SIC register and PC to see if it matches the next system call record in the trace file. Because all other aspects of the replay are correct, we will execute system calls in the same order as the original execution, so we only need to check the data at the current position in the trace file. If the system call matches the trace data, we restore all of the traced values to the user address space, and advance to the next system call record for the current interval in the trace file.

7.2 Replaying Interrupts

The replay tool must also guarantee correct replay of software interrupts, including the interrupt which marks the end of the interval. As discussed earlier, these two cases can be handled identically since our interval hook is triggered by a software interrupt generated by the UNIX interval timer, and we record the *(SIC, PC)* pair, UNIX signal number, and signal handler disposition for all interrupts. To correctly replay interrupts, we read the traced information for interrupts prior to the re-execution of each interval, and patch the instruction at each program counter location where an interrupt occurred with an instruction to generate a software trap. In our trap handler, we check the value of the SIC register to see if it matches the traced SIC value for the interrupt which occurred at this PC value. If it does, we restore the signal handler disposition and raise the appropriate signal to replay the interrupt. Otherwise we execute the code replaced by the patch, and continue the replay.

8 Future Work

In the area of DFA design and implementation, several issues merit further consideration. Given a series of constraints such as the rules which define a unique-spanning read, it may be possible to devise an algorithm which can produce a DFA with an optimal bit layout. Alternatively, given a DFA design, we would like to develop an algorithm which produces the optimal bit layout. These theoretical issues may allow us to develop DFAs for tracing which require less run-time overhead.

It may also be possible to provide new or more efficient types of debugging facilities using the DFA techniques described in this work. Other applications may include state machines which record a (SIC, PC) pair marking the most recent modification of a given address, allowing debugging queries for the last instruction to modify a particular variable, or more efficient implementations of memory access checking. We hope to explore these applications as part of our ongoing research.

To complete the research described in this paper, a prototype replay tool will be developed. The user interface to an incremental replay tool is itself an issue for further research. We propose to explore the types of debugging queries a replay tool can answer, and how the user can specify which intervals of the program to replay. Finally, we propose to investigate the use of our trace and replay tool to perform flowback debugging analysis. We hope to develop debugging tools which allow users to identify incorrect program state, query the last modification of a given variable, and then replay the modification to this data to determine the source of bugs.

References

- [1] Stuart I. Feldman and Channing B. 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.
- [2] Robert Garner. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., New Jersey, 1992.
- [3] J.M. Mellor-Crummey and T.J. LeBlanc. A software instruction counter. In *Proceedings of the Third ASPLOS*, April 1989.
- [4] Robert H.B. Netzer. Personal communication, October 1995.
- [5] Robert H.B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–325, Orlando, FL, June 1994.
- [6] Sun Microsystems, Inc., Mountain View, CA. *Linker and Libraries*, 1994.
- [7] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, Inc., New Jersey, 1996.