Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs

Robert H.B. Netzer and Mark H. Weaver

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-11
March 1994

Appears in the SIGPLAN Conference on Programming Language Design and Implementation, June 1994

Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs

Robert H. B. Netzer *Mark H. Weaver*
r n@cs.brown.edu Mark_Weaver@brown.edu

Dept. of Computer Science
Brown University
Box 1910
Providence, RI 02912

1. Introduction

Debugging requires execution replay. Locations of bugs are rarely known in advance, so an execution must be repeated over and over to track down bugs. A problem arises with repeated reexecution for long-running programs and programs that have complex interactions with their environment. Replaying long-running programs from the start incurs too much delay. Replaying programs that interact with their environment requires the difficult (and sometimes impossible) task of exactly reproducing this environment (such as the connections over a one-day period to an X server). These problems can be solved by periodically checkpointing the execution's state so it can be *incrementally* replayed or restarted from intermediate points. Restarting from checkpoints bounds the delay to replay any part of the execution if checkpoints were taken often enough, and allows parts of the execution having interactions with the environment difficult to reproduce to be skipped. However, the time and space costs of such checkpointing can be prohibitive. To reduce this cost, we present *adaptive* tracing strategies that provide bounded-time incremental replay and that are nearly optimal. Our implementation on a Sparc 10 traces less than 15 kilobytes/sec for CPU-intensive programs and for interactive programs the slowdown is low enough that tracing can be left on all the time.

Our first result is a tracing strategy that adaptively decides *when* and *what* to trace to provide bounded-time incremental replay. To decide when to trace, we divide the execution into *windows*, which are regions of the execution we can individually replay. To decide what to trace, we observe that a window's correct reexecution requires each of its reads to return the same value during replay as during the original execution. Each such value *need not be traced* if the read is preceded in the same window by a write to the same location (the write will restore during replay the original value) or if the value was already traced. To bound the replay time, we trace reads often enough so their values are never very far back in the trace, allowing the state required for the replay to be quickly restored. We also place a maximum length on the windows. The optimal tracing problem is to adaptively decide where to start windows, and what memory references in each to trace, to minimize the number of references traced. We show that the problem can be reduced to computing the shortest path through a directed graph, solvable in $O(nT)$ time (where n is the number of member references made during execution, and T is the maximum number of references allowed in a window).

Although the optimal problem is solvable, it is probably impractical as it seems to require more than constant work at each memory reference. For efficient tracing, we present approximations that employ *fixed-size* windows. These approximations are simple and efficient. They perform simple bitvector lookups at each read and write using space-efficient two-level bitvectors. Experiments show they usually trace at most twice more than optimal. Our implementation on a Sparc 10 incurs a factor of $1.75-7$ slowdown and generates less than 15 kilobytes/sec of trace on CPU-intensive programs. In addition, we instrument system calls to automatically trace only those interactions a program has with its environment that affect its outcome. This feature addresses one of the difficult aspects of replay, allowing us to trace and replay interactive systems.

Our work is novel in that our tracing algorithms adapt to the particular execution being debugged. A difficult part of supporting incremental replay is bounding the replay time while tracing only what is necessary. We achieve both of these objectives. Previous systems that provide incremental replay can trace more than necessary and can incur large delays during replay[5, 10, 4, 3]. We guarantee a user-specified bound on the time of any replay by fixing the size of windows and by tracing often enough.

Adaptive tracing is becoming not only practical but necessary. In the past, the cost of disk I/O was insignificant compared to even a small amount of computation (such as the bitvector manipulation our algorithms perform). Now, processors are so fast that a significant work can be performed in the time it takes to write only a single trace record. Since this trend is continuing, in the future it will be cheaper to use adaptive strategies that do on-line analysis to minimize tracing than to trace more than necessary.

This research was partly supported by ONR Contract N00014-91-J-4052 (ARPA Order 8225) and NSF grant CCR-9309311.

2. Related Work

An execution must be traced to provide incremental replay. Enough must be recorded so that during replay sufficient state can be restored for each read from memory to obtain the same value as during the original execution. Below we describe past approaches to this problem[5, 10, 4, 3, 1]. These approaches either trace orders of magnitude more than necessary (as shown later), or do not bound the time required to replay up to an arbitrary point of the execution.

The IGOR system uses the virtual memory system to periodically trace, at fixed time intervals, those pages modified since the last checkpoint[5]. To restart the execution from an intermediate point requires scanning the trace to find the most recent checkpoint of each page. Because checkpoints are taken at fixed time intervals, IGOR bounds the amount time required to replay any part of the execution *once replay begins*. However, setting up the state for replay requires potentially scanning through the entire trace file, which can take time proportional to the length of the execution. Although this approach is adaptive in the sense that it traces only pages that have been recently written, our experiments show that a page granularity of tracing is too large to be practical; orders of magnitude more data than necessary is traced.

The PPD system uses compile-time analysis to decide what and when to trace[10, 4, 3]. PPD writes a *prelog* on the entry of each procedure, containing the variables the procedure might possibly read before defining. The prelog allows a procedure to be re-executed in isolation since it contains all variables the procedure might read. A *postlog* is written on procedure exit, containing the variables the procedure might have modified. The postlog allows the execution of a procedure to be skipped during replay since it contains the changes the procedure might make to the state. PPD has the drawback of statically computing what variables are traced and when they are traced. Since compile-time analyses must be conservative, more variables can be placed in the logs than necessary. In addition, tracing only at procedure entry and exit can sometimes incur high overhead and provide no guarantees on replay time. For example, a 1000-iteration loop that contains a procedure call incurs many needless traces[4], and a very long-running procedure may not be traced often enough to replay any part of it in a reasonable time. As an attempt to alleviate these problems, prelogs and postlogs can optionally be generated for loops, and not generated for some procedures[4], but the basic idea is limited by its static nature.

The Spyder system traces, before each statement (or group of statements), its *change set*, the values of the variables the statement might modify[1]. A debugger can *backup* execution over a statement by restoring the state from its change set. As an optimization to bound the trace size, only the most recent change set from each statement can be kept. Spyder statically computes the change sets, and for programs that use pointers and arrays, it must trace each such access. Spyder's main disadvantage is that it does not bound the time required to perform a replay, since we must first back up to before the desired interval and then reexecute forward. Although Spyder provides a valuable backup tool, like PPD it is limited by its static nature (and must trace every array or pointer reference), and it was not designed to provide a general replay facility that can quickly reexecute any requested part of the execution.

Wilson proposes an idea called *Demonic Memory* as a way to recreate any past state of a process[14]. By maintaining a hierarchy of checkpoints, each taken at successively larger time granularities, recent states can be reproduced quickly while older states incur more delay. He proposes *virtual snapshots* as a way to checkpoint, where only the parts of a checkpoint that differ from the previous one are saved. Our techniques are complementary to Demonic Memory as they provide another way to take the checkpoints, and a hierarchy of different window sizes can be used to compact our trace on-the-fly.

Checkpointing and tracing strategies have been proposed for problems such as fault tolerance, profiling, and others[8, 11, 2, 7, 6]. The incremental replay problem is more difficult than these problems since user replay requests occur often (unlike faults, for example) and they must complete quickly. In addition, unlike performance measurement, the tracing requirements are substantial, since enough state must be recorded to make the replay identical to the traced execution. Tracing strategies for these other problems therefore do not solve the incremental replay problem.

3. Adaptive Tracing for Bounded-Time Replay

Past techniques are limited by their static nature, deciding at compile-time what or when to trace (or both). As a result, they trace more than absolutely necessary and may incur lengthy delays in replaying up to a requested part of the execution. We overcome these problems by *adaptively* deciding what and when to trace. To decide when to trace, we divide the execution into *windows* — contiguous intervals of the execution that can be individually replayed — and produce a set of traces for each window. To decide what to trace, we perform on-line analysis at each memory reference to dynamically determine which to record. Below we outline our adaptive approach. We show that optimally deciding what and when to trace has polynomial time complexity, but is too expensive to be practical. In the next section we present nearly optimal approximations that are efficient.

Providing bounded-time replay involves two considerations. First, the time it takes to setup the state (from the trace) for any replay must be bounded. We bound this time by tracing often enough so that restoring the state any window will need requires reading at most M previous trace records. Second, the time it takes to replay up to any part of the execution, once replay begins, must be bounded. We bound this time by placing a limit of T memory references on the length of any window. There is a tradeoff between fast replay and small traces; larger values of T and M result in smaller traces but longer replay times. In practice we expect the user to provide T and M to tune this run-time *vs* replay-time overhead tradeoff.

3.1. What to Trace

To support the replay of some portion of the execution (a window), we wish to trace just enough to ensure that a reexecution from the window's start correctly reproduces the original. We consider a replay correct iff each read from memory receives the same value as during the original execution. We also wish to trace often enough so the state necessary for the replay can be restored from the trace in bounded time. We consider a window's replay to be *M-bounded* iff we need to scan at most the M previous trace records before the window's trace to restore its state. The following proposition shows what must be traced.

Proposition 1

> In general, enough trace data exists to provide an M-bounded replay of a window iff for every read in the window, either
>
> (1) the value read is saved in one of the M trace records preceding the start of the window's trace, or
> (2) a write to the same memory location is previously made within the same window.

If a memory location being read was already written in the same window, there is no need to save its value; during replay the preceding write will restore the original value. If a location is read in a window but its value was recently traced (within the current window or last M trace records before the window), it need not be traced again; to replay, the value can be retrieved without scanning too far back in the trace file. For M-bounded replay, we only need to trace reads that would violate the above two conditions if not traced. We call such reads *unique-spanning$_M$* reads. A unique-spanning$_M$ read is the first read from an address A in a window where A was written in an earlier window but its value not traced in the M trace records recorded before the window.

Without knowing the semantics of a window's computation, tracing unique-spanning$_M$ reads is both necessary and sufficient for supporting M-bounded incremental replay. If we do not trace the value of a unique-spanning$_M$ read, then there would exist a window which, when reexecuted in isolation, would contain some read not preceded by a write to the same address, nor saved in last M trace records. We would be unable to start this window's replay by reading at most M trace records.

An alternative for supporting incremental replay is to instead trace the *writes* to memory. There are tradeoffs between tracing reads and writes, which we discuss later. If we define a *unique write* as the last write to each location in a window, then tracing the unique writes in each window provides sufficient information for ∞-bounded replay (i.e., no bound can be guaranteed). No bound can be guaranteed because if we trace only the writes, we do not know what addresses a window's replay will read, so we have to restore the state from the traces of all preceding windows. As discussed in Section 2, this potentially requires scanning through the entire trace. We explore this alternative because (as we show later) less run-time overhead is incurred in locating unique writes than unique-spanning reads, so they provide an alternative when run-time overhead must be minimized. Figure 1 shows examples of unique-spanning reads and unique writes.

3.2. When to Trace

Deciding the second part of adaptive tracing, *when* to trace, involves determining when to start a new window. We wish to adaptively start a new window subject to two constraints: (1) the number of references traced is minimized, and (2) no window is longer than T references, the user-specified bound on the maximum allowable replay time. We can minimize the number of references traced by placing windows carefully, for example to group together as many references to the same address as possible. By keeping windows no longer than T references, the time to actually reexecute up to any part of any window remains bounded.

A surprising result is that computing where to start new windows to minimize the number of unique-spanning$_M$ reads traced is not NP-complete for $M = 0$ or $M = \infty$. Determining optimally where to start each window can be performed in $O(nT)$ time, where n is the number of memory references. Although we omit the proof details for brevity, the basic idea is that determining when

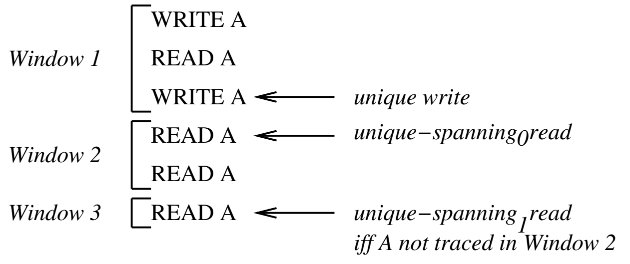


Figure 1. Example execution broken into windows.

to start new windows can be reduced to computing the least-cost path through a directed graph. Each memory reference is a node in the graph. Each node has edges to nodes representing each of the subsequent T references, with each edge labeled with the cost of tracing (the number of unique-spanning$_M$ reads) if the two connected nodes were to delimit a window. A least-cost path is found in time $O(nT)$ because each of the n nodes has degree T (the head of each edge in this path shows where each window should be started). We are unsure of the complexity for other values of M and leave an investigation to future work. However, because the cases $M = 0$ and $M = \infty$ perform well (discussed later), it is unclear whether other values of M are of practical interest.

4. Fixed-Window Tracing Algorithms

Even though optimally deciding *when* to start a new window has polynomial time complexity, it seems to require more than constant work at each memory reference, which is too inefficient to run on-line. We overcome this problem with an approximation by simply fixing each window size at T. The problem then reduces to adaptively deciding which references to trace in each window. Below we conceptually outline tracing algorithms for providing M-bounded replay for any M, and for the special cases of $M = 0$ and $M = \infty$ which have simple algorithms. We also present details of an implementation for the Sparc.

When $M = 0$ we trace enough so that a window can be replayed by simply restoring the state from its own trace. Other parts of the trace need not be scanned, providing the lowest replay setup time. When $M = \infty$ we do not worry about how far back in the trace we must scan. This allows fewer reads to be traced at the expense of longer replay time, as there are no more unique-spanning$_\infty$ reads than unique-spanning$_M$ reads for any M. We present these two special cases since they have simple implementations and are probably of the most practical interest.

4.1. Conceptual Algorithms

Figure 2 shows our algorithms. An action is performed when a window boundary is reached (the *window_boundary_hook* function) and after each read or write to memory (*read_hook* and *write_hook*, where the parameter a is the address being accessed). The special case of tracing unique-spanning$_\infty$ reads works as follows. *Traced* is a bitvector that indicates which addresses have been traced since last written, and the *NeedToTrace* bitvector marks which addresses have not been written in the current window or traced since last written. When a write occurs to address a, we reset the a^{th} bit in *NeedToTrace* (indicating that a has been written in the current window so a subsequent read in the same window need not be traced) and in *Traced* (indicating that a new value has been written into a which has not yet been traced). When a read from a occurs, we check to see if it is unique-spanning$_\infty$: if the value in address a has not been traced since last written *or* written yet in the window, *NeedToTrace*[a] will be 1 and the read's address and value are traced. Otherwise, the read is not unique-spanning$_\infty$ and can be ignored. At the start of each window we set the *NeedToTrace* bitvector as the complement of *Traced* since any addresses traced since last written need not be traced again.

We generalize this algorithm to detect unique-spanning$_M$ reads as follows. By changing *Traced* into a counter, we track how many *windows* elapse before a traced value ‘‘expires’’ in the sense that it must be traced again if read. *Traced* only need be large enough to store values between 0 and an expiration count, C, and only the number of windows since a value is traced need be counted. In the implementation we could count the actual number of trace records (e.g., by counting the number of times a trace buffer is flushed to disk), but for simplicity we count windows. Any read not previously written in the same window is traced if its value was not traced within the last C windows. At each window boundary we decrement the *Traced* counter associated with each address and set the *NeedToTrace* bit when the count reaches 0.

The algorithms to trace unique writes and unique-spanning$_0$ reads are similar. To detect unique-spanning$_0$ reads we simply trace the first read to an address in a window if it was not previously written in that window, so a counter is unnecessary.

Since most memory references are four-byte accesses, as a bitvector optimization we need not assign one bit per address but only one bit per word. Sub-word references are handled as shown in Figure 3. If we are tracing unique writes, a sub-word write is treated as a write to the entire word. If we are tracing unique-spanning reads, we must handle sub-word writes by resetting *Traced* to 0, ensuring that subsequent reads to any part of the word will be traced if not already written in the window. When tracing unique-spanning$_0$ reads, the *Traced* counter is not used, so the sub-word hook becomes empty.

An advantage of tracking reads and writes on-line is that the values a window's replay will need (and that will not be recomputed during replay) are *automatically* traced. We can extend this feature of the algorithm to handle one of the challenging aspects of replay: programs that have interactions with their environment. If we specially treat system calls, we need not reexecute them during replay, and any interactions a program has with its external environment through these calls can automatically be reproduced.

We instrument each system call to perform several actions (*sys_call_hook* in Figure 3), although we need not track each read and write it makes. First, we estimate which addresses might possibly be written during the call. Estimating these addresses is usually straightforward since

```
window_boundary_hook {
    trace addresses & values of
        locations in Written;
    Written bit-vector = 0's;
}
```

```
(no read hook required)
```

```
write_hook(a) {
    Written[a]=1;
}
```

(a): unique writes

```
window_boundary_hook {
    foreach a
        if ( Traced[a] > 0 )
            NeedToTrace[a]=0;
            Traced[a]−−;
        else
            NeedToTrace[a]=1;
}
```

```
read_hook(a) {
    if ( NeedToTrace[a] )
        trace address & value;
        NeedToTrace[a]=0;
        Traced[a]=C;
}
```

```
write_hook(a) {
    NeedToTrace[a]=0;
    Traced[a]=0;
}
```

(b): unique-spanning$_M$ reads

```
window_boundary_hook {
    NeedToTrace bit-vector = 1's;
}
```

```
read_hook(a) {
    if ( NeedToTrace[a] )
        trace address & value;
        NeedToTrace[a]=0;
}
```

```
write_hook(a) {
    NeedToTrace[a]=0;
}
```

(c): unique-spanning$_0$ reads

```
window_boundary_hook {
    NeedToTrace bit-vector =
        NOT ( Traced bit-vector );
}
```

```
read_hook(a) {
    if ( NeedToTrace[a] )
        trace address & value;
        NeedToTrace[a]=0;
        Traced[a]=1;
}
```

```
write_hook(a) {
    NeedToTrace[a]=0;
    Traced[a]=0;
}
```

(d): unique-spanning$_\infty$ reads

Figure 2. Fixed-window-size tracing algorithms, performed at each window boundary (window_boundary_hook), each read (read_hook) and each write (write_hook).

most system calls access contiguous regions of memory (such as reading from disk into a buffer).

Second, after the system call we must either specially update the bitvectors or emit a trace, depending on the type of tracing being done. If we are tracing unique writes, we must immediately trace the values of the locations possibly modified. We cannot wait until the window's end since the values might be read or written any time after the call. We must also zero the associated bits in *Written* so the locations will not be re-traced at the window's end unless they are subsequently written. However, if we are tracing unique-spanning reads, we can fool the algorithm into automatically tracing these values at the point where they are subsequently read. By resetting bitvector entries corresponding to the modified locations, any subsequent reads of these locations will be detected as unique-spanning and traced as usual. We could immediately trace the values as done when tracing unique writes, but resetting the bits has the advantage that values not subsequently used will not be traced.

Finally, we always emit a trace record (not shown in Figure 3) indicating that a system call was made, but need not record which one since exactly the same call will be issued during replay. As an optimization, if we know that reexecuting a particular system call will exactly reproduce its original execution (e.g., a read from a read-only file), we can treat it as any other sequence of reads and writes.

This scheme fails for interactions with the environment where state is maintained outside of the user's process (such as on a bit-mapped display). When state changes to an external device must be reproduced, the device's state must also be traced during execution and restored during replay when the system call is reached. Instrumentation must be added to the system call to handle the details of how the device's state is accessed.

We can also integrate into our scheme an approach for tracing when interrupts occur and producing them during replay[9]. By instrumenting the program to maintain a software instruction counter, interrupt handlers can trace the value of the counter and the PC when an interrupt occurs. During replay a watchpoint can be set for the location at which the interrupt initially occurred to check the instruction counter against the traced value. When the counter reaches the traced value, a jump can be made to the interrupt handler.

We finally mention an attractive property of the algorithm: it can gracefully degrade in the presence of limited disk space for traces. If trace space is limited, the trace (or parts of it) can be compacted on-the-fly by suspending execution and compressing the trace file generated so far. By re-running the tracing algorithm *from the trace*, and

---

(no sub-word read hook required)

```
subword_write_hook(a) {
    w = addr of word containing a;
    Written[w]=1;
}
```

```
sys_call_hook() {
    make the system call;
    estimate addresses written;
    for all written locations, a
        trace a & value at a;
        Written[a] = 0;
}
```

(a): unique writes

```
subword_read_hook(a) {
    w = addr of word containing a;
    call regular read_hook(w);
}
```

```
subword_write_hook(a) {
    w = addr of word containing a;
    Traced[w] = 0;
}
```

```
sys_call_hook {
    make the system call;
    estimate addresses written;
    for all written locations, a
        Traced[a] = 0;
        NeedToTrace[a] = 1;
}
```

(b): unique-spanning$_M$ reads

```
subword_read_hook(a) {
    w = addr of word containing a;
    call regular read_hook(w);
}
```

(no sub-word write hook required)

```
sys_call_hook() {
    make the system call;
    estimate addresses written;
    for all written locations, a
        NeedToTrace[a] = 1;
}
```

(c): unique-spanning$_0$ reads

```
subword_read_hook(a) {
    w = addr of word containing a;
    call regular read_hook(w);
}
```

```
subword_write_hook(a) {
    w = addr of word containing a;
    Traced[w] = 0;
}
```

```
sys_call_hook() {
    make the system call;
    estimate addresses written;
    for all written locations, a
        Traced[a] = 0;
        NeedToTrace[a] = 1;
}
```

(d): unique-spanning$_\infty$ reads

Figure 3. Hooks for tracing partial-word reads and writes, and system calls.

increasing T or M, a new trace file is produced smaller than the original. For example, doubling T will double the replay bound but reduce the trace size. This doubling can be performed indefinitely and will in the limit will reduce the trace to only the reads of uninitialized variables and the values traced from system calls (with T at its maximum, only these reads are unique-spanning). Thus, executions of great length can probably be traced, with T or M doubling automatically as needed.

4.2. Implementation on the Sparc

The tracing algorithms are conceptually simple but because they perform on-line analysis at each read and write operation they require careful implementation. We have developed several strategies to instrument programs at the assembly level to keep the run-time overhead acceptable. We focus on the Sparc, but the general ideas are applicable to any RISC processor. Although the instrumentation could also be added by a compiler, by editing the executable to jump to an appropriate hook after every memory reference[12, 7], or by hardware support, our implementation is tuned for our particular application.

To make the bitvector operations efficient, we employ *two-level* bitvectors and maintain one bit per four-byte word as discussed above. Since we do not know in advance which addresses the program may reference, a flat bitvector for a 32-bit address space would occupy 128 megabytes of storage. A two-level bitvector is a table of pointers to bitvector fragments, with each fragment being allocated only when a reference is made to the memory region it represents. Accessing them is still quick, requiring only two memory references, and we show in Section 6 that their space overhead is low. Other work has also reported good experience with two-level bitvectors[13]. In addition, assigning only one bit per word leaves the bottom two bits of each address to be used as tags to encode the trace record type (to indicate a value, system call, signal, or end-of-window marker).

Since the bitvector operations must be performed at each read or write, we optimize them for the common case. The common case is a read that is not unique-spanning and thus not traced (Section 6 shows that only a small percentage of reads require tracing). Since writes occur less frequently than reads, the performance of the write hook is less critical (except when tracing unique writes, which requires no read hook). A crucial part of the instrumentation is that condition codes must not be altered since instrumentation can be inserted anywhere a load or store occurs. However, there is no user-level instruction on the Sparc to save condition codes, although an (expensive) operating system trap exists to save them. We overcome this problem by coding the instrumentation so it does *not* alter the condition codes in the common case. Even though the instrumentation must do several checks, such as determining whether a bitvector segment is currently allocated or whether the bit itself is zero, we can perform these checks without altering the condition codes. The idea is to do an *indirect* jump to an address computed from the value of the bit being tested, thereby avoiding a traditional conditional jump (which would require altering the condition codes). The jump either returns from the instrumentation or calls another routine to perform additional work (if a bitvector segment must be allocated or if a read requires tracing).

Another important trick we employ is to implement a software instruction counter (SIC) to provide the ability to replay interrupts and avoid doing end-of-window checks at each load and store instruction. We adapt an idea proposed by Mellor-Crummey[9] of incrementing a counter at each backward branch and procedure call. The value of this counter together with the PC uniquely identifies each instruction instance. We could use such a counter to determine when a window boundary is reached, but if such a check were done before each backward branch, the condition codes would have to be saved. To avoid this overhead, we instead increment and test the counter before each instruction that alters the condition codes so they need not be saved. Since there is probably one such instruction for each conditional branch, such a check still locates window boundaries with reasonable accuracy. Another way to determine when a window boundary is reached is to employ an interval timer that causes a signal handler to be called periodically (with the period specified by the user). The handler patches one word in the common part of the instrumentation code (which every read or write hook calls) so that on the next memory reference the window boundary processing is performed (*window_boundary_hook* in Figure 2) and the code is patched back to its original state. This scheme avoids doing a test each time the SIC is incremented, but requires that an interval timer be available.

Since we expand each assembly instruction that references memory into a sequence of instructions, we must also carefully handle instruction delay slots (such as those for branches). We move the instrumented instruction out of the delay slot to before the branch, or move one copy to each target of the branch. Correctly handling delay slots in all cases is actually more complex, but we omit the details here.

5. Replaying from the Trace

We now discuss how to replay any of the execution's windows. There are two parts to replay: restoring the initial state of the window's memory from the trace, and performing the replay itself. An important point is that the *same* instrumented code must be used during reexecution as during the original execution. Otherwise, if a different (uninstrumented) version of the program were reexecuted, any function pointers restored from the trace might not have the proper value, although the hooks can be changed to perform replay-specific tasks as long as their size remains unchanged.

5.1. Restoring the State

To replay a given window, we must restore some of the memory's state (and all the registers) before reexecution. This state is restored differently depending on what type of tracing was performed. When unique-spanning$_M$ read tracing was done, the values that the window will need (and that will not be recomputed during replay) are guaranteed to be in the window's traces or the traces of the previous M/T windows. We simply restore the state by scanning the previous M/T windows' traces, and the current window's trace but *only* up to the trace of the first system call. Traces made after a system call must not be restored until after the call is skipped during replay (we address this issue below). Restoring this state is straightforward: we make a linear pass through the trace and place into memory the value in each trace record.

Restoring the state when unique write tracing was done is more complex. For unique writes, we do not know which memory locations will be read by the window's replay (this information was not initially traced). We must therefore restore the entire state of memory which appears in the trace before the window's traces. System calls are handled as above. With the exception of handling system calls, this scheme is identical to how the IGOR system (discussed in Section 2) restores its state, and might be made easier by perhaps constructing indexes to identify the most recent trace of each address. An alternative is to trace the addresses that each window actually reads and restore only these locations, but such an approach would incur more run-time overhead and still require scanning the trace to find the necessary trace records.

5.2. Handling the Replay

Once the state is restored, the reexecution begins by jumping to the appropriate location in the program (we assume the program counter was traced at each window boundary). Once reexecution begins, we must also specially handle system calls.

We instrument system calls so they know that a replay is in progress and the call into the system is not made. Instead, additional state is restored before returning to reproduce the side-effects of the call. We read and restore values from the trace file from the point last read up to the trace record for the next system call (or to the window's end, whichever occurs first). This restores sufficient state to replay up to the next system call, since during execution any values possibly modified by the call were either traced when the call returned or traced when eventually read. Once these values are restored, the program can continue reexecuting.

6. Experimental Measurements

We performed four experiments to assess our ideas. First, we compared the tracing of unique writes, unique-spanning$_0$ reads, and unique-spanning$_\infty$ reads. Second, we compared trace sizes produced by our fixed-window size algorithms to the optimal algorithms. The fixed-window algorithms are effective, producing traces less than 50% larger than optimal. Third, we compared our approaches to the IGOR[5] and PPD[4, 10] systems. We trace $1-2$ orders of magnitude less than IGOR, and usually $4-50$ times less than PPD when PPD provides quick replay. In some cases the PPD traces are small but they cannot provide quick replay, and in these cases we provide quick replay with a trace of less than twice as large. Finally, we measured the run-time overhead of an implementation of our algorithms on a Sparc 10. Our algorithms incur slowdowns of about a factor of $1.75-7$ (depending on which type of tracing is done) and generate about 15 kilobytes of trace per second. We conclude that our scheme simultaneously bounds the replay time while keeping trace generation low, and is probably faster than writing large traces to relatively slow disks.

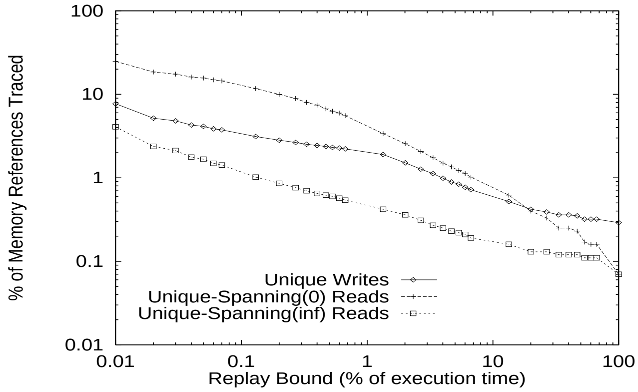
We divided our experiments into two parts: one to analyze the trace sizes and another to measure run-time overhead. To analyze the trace sizes, we ran four programs using the *vmon* tool which rewrites executables to obtain the necessary hooks[12]. This allowed us to call a function at each memory reference that simulated our tracing algorithm on all possible window sizes simultaneously. We analyzed the C compiler supplied with SunOS version 4.1.3 (*ccom*) when compiling an 11,214-line program, the UNIX compress utility (*compress*) when compressing the *lex* executable, a program that randomly generates directed graphs and runs two scheduling algorithms on the graph to compare their results (*event*), and a program that computes finite differences by making one pass over a 200×200 mesh (*mesh*). The ccom, compress, and event programs represent computations of a symbolic nature; mesh is numerical. We analyzed other programs as well, but the trace sizes of these four are representative. The *ccom*, *compress*, *event*, and *mesh* programs performed 15 million, 1.34 million, 122 thousand, and 539 thousand memory references, respectively.

To analyze run-time overhead, we wrote an assembly level instrumentor using the techniques outlined in Section 4 and ran several long-running programs (discussed later).

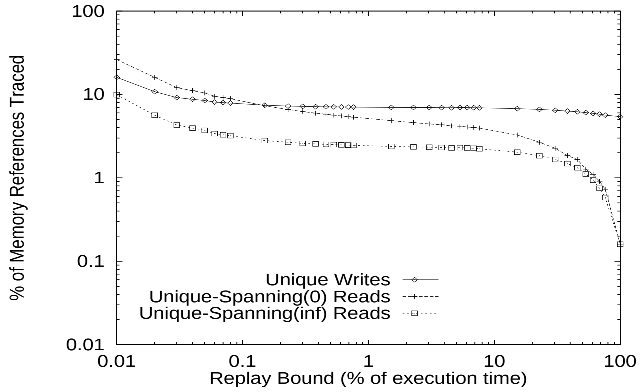
The curves in this section plot the replay bound (i.e., the fixed window size T) *vs* trace size. Since the programs had different execution lengths, the replay bound and trace size are given as a percentage of the total number of memory references. Due to the large difference between some of the curves, the plots appear on log-log scales.

6.1. Comparison of the Three Types of Tracing

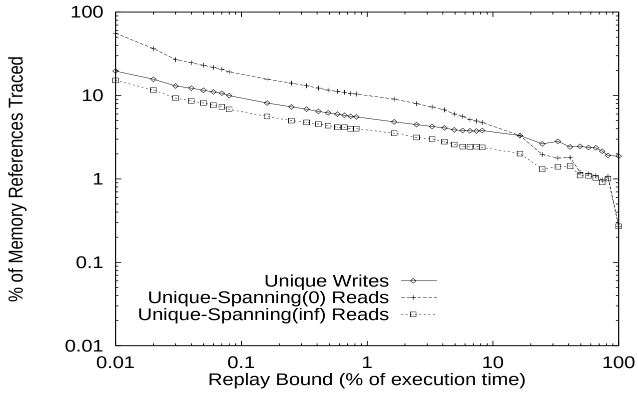
Figure 4 compares for each program the trace sizes of unique writes, unique-spanning$_0$ reads, and unique-



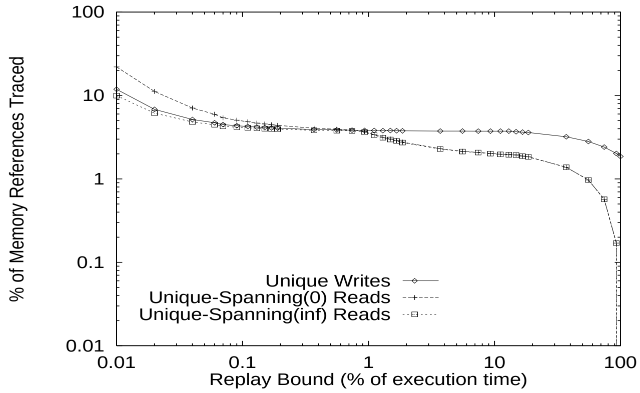
(a): ccom



(b): compress



(c): event



(d): mesh

Figure 4. Comparison of the three styles of adaptive tracing.

spanning$_\infty$ reads. There are $3-100$ times fewer unique-spanning$_\infty$ reads than other types of operations, and unique writes did as well or better than unique-spanning$_0$ reads. Locality of reference means that many reads are often preceded by writes in the same window and thus are *not* unique-spanning (but the writes are still unique writes). However, in some cases (notably *ccom*), data that is written once and read many times (such as the compiler's symbol table) causes fewer unique writes to exist than unique-spanning$_0$ reads. We conclude that unique-spanning$_\infty$ reads are preferably when trace size must be reduced as much as possible. Otherwise, the run-time *vs* replay-time overhead tradeoff must be considered; unique-spanning$_0$ reads guarantee bounded replay time, but detecting unique writes incurs less run-time overhead (discussed later). Note that the trace sizes often tend to decrease exponentially with increasing window sizes, suggesting that a window size as large as tolerable should be used.

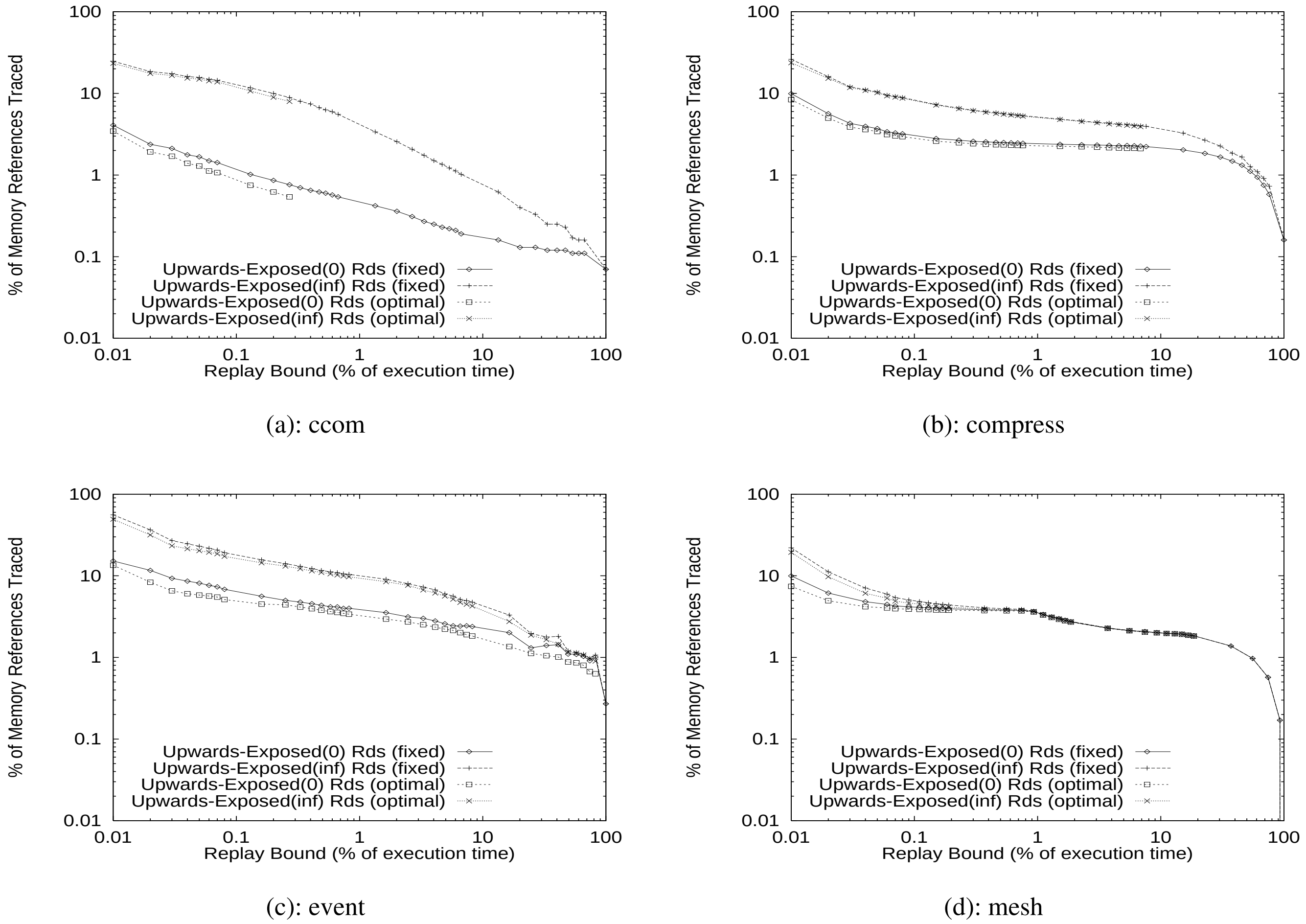
6.2. Comparison of Fixed-Window and Optimal Tracing

Figure 5 compares the trace sizes for the fixed-window algorithm and the optimal algorithm for unique-spanning$_0$ and unique-spanning$_\infty$ reads (the fixed-window algorithm for unique writes is already optimal). By "optimal" we mean that the fewest number of unique-spanning reads are traced (under the assumptions discussed in Section 3). We implemented the optimal algorithm by constructing a weighted directed graph on-line and computing the shortest-path through the graph. The fixed-window algorithms generated traces at most 50% larger than optimal for *ccom* and *event*. For *compress* and *mesh*, the algorithms were nearly optimal. Because this analysis required $O(nT)$ time (where n is the number of memory references and T is the maximum window size) it became expensive as T grew and for *ccom* (the longest running program) optimal data points beyond $T = 0.3\%$ could not be obtained. Fixing the window size appears to work well because of locality: for almost any window placement, most reads inside the window are not unique-spanning. Thus, fixed windows are nearly as good as carefully placed, optimal windows.

6.3. Comparison to IGOR and PPD

Figure 6 compares the size of our adaptive traces to traces generated by the IGOR and PPD systems. IGOR checkpoints periodically, such as every *S* seconds (where *S* is given by the user), although we checkpointed every *T* memory references for this study. Each checkpoint saves to disk the pages written since the last checkpoint (our page size was 4k bytes)[5]. However, as discussed in Section 2, IGOR does not guarantee how long it can take to restore the state for a replay since the entire trace file might need to be read. IGOR wrote huge traces, over two orders of magnitude larger than the unique-spanning read trace. Despite locality, a page granularity of tracing is too large to keep the trace size small, especially when *T* is small (i.e., when the user needs quick replay). The IGOR curves in Figure 6 go off the top of the scale and are not completely shown.

PPD generates a prelog and postlog for each procedure called. We simulated PPD tracing by computing the *exact* logs. A real PPD system uses compile-time analyses to conservatively determine which variables should be in the log, and traces every access to variables it is unsure about (such as references through pointers). Our simulation thus underestimates the PPD trace size. Since checkpoints are not written at regular intervals, PPD does not guarantee how long a replay may take. Note that the PPD curves are flat in Figure 6 since *T* is not a parameter to PPD (but we still show PPD's trace size on the figure for comparison).



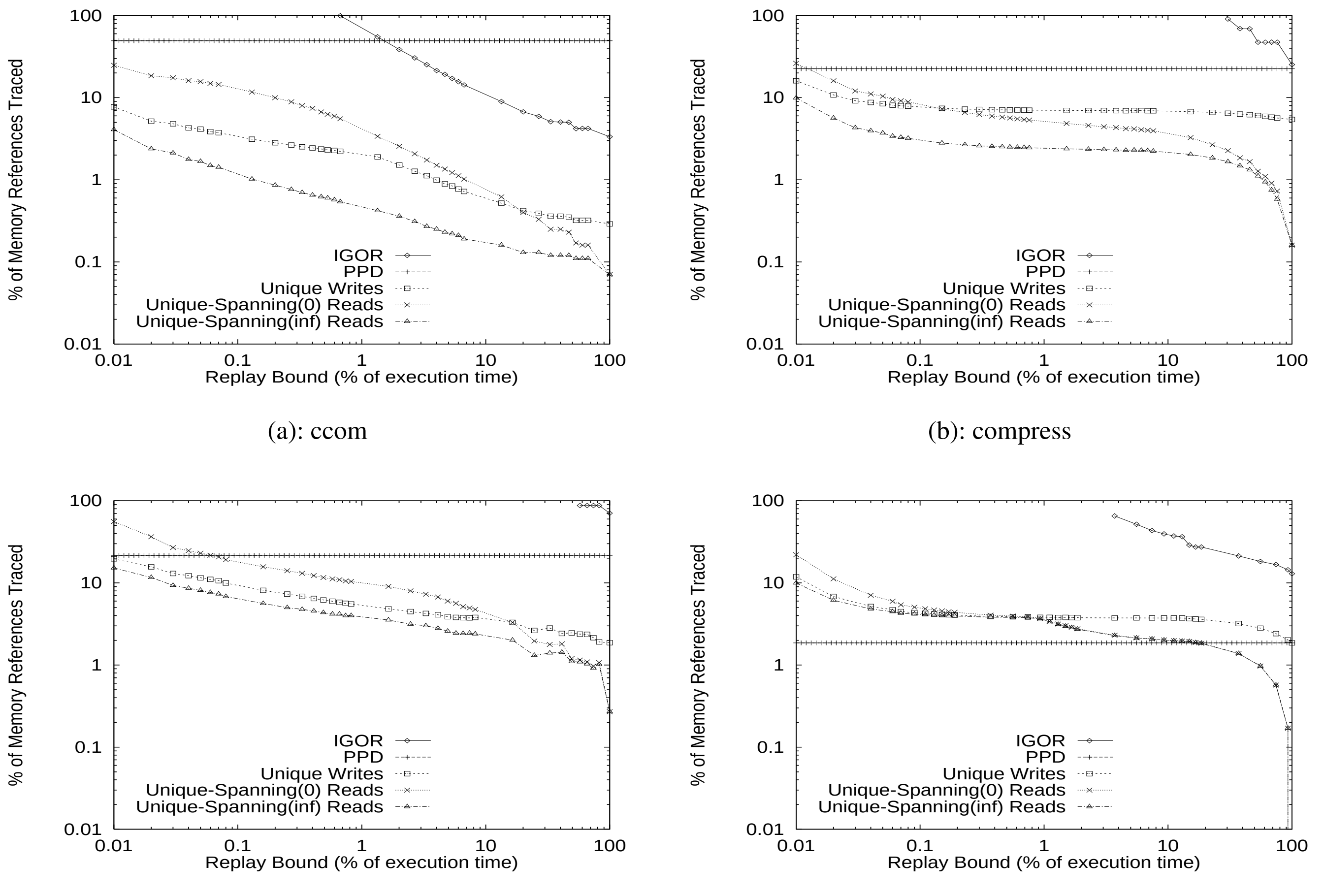
(a): ccom

(b): compress

(c): event

(d): mesh

Figure 5. Comparison of fixed-window and optimal tracing.



(a): ccom

(b): compress

(c): event

(d): mesh

Figure 6. Comparison of trace sizes to other systems.

To make a fair comparison, we must consider this issue. For *ccom* and *compress*, the PPD traces were sufficient to provide a replay bound of about 0. 1% of the execution (i.e., most procedures lasted about 0.1% of the total execution time). For *event* the bound was between 0. 1 and 17%, and for *mesh* the bound was 99%. *Mesh* consists of one large procedure, and since PPD writes a trace only on procedure entry and exit, it is unable to start the replay from inside the procedure. Figure 6 shows that when PPD provides quick replay, we can achieve the same replay bound with traces $4-50$ times smaller. For *mesh* (where PPD provides slow replay), we can provide quick replay while tracing only $0-2$ times more. These results show that adaptive tracing with fixed-size windows provides bounded replay while recording only a few percent of the memory references.

6.4. Run-Time Overhead

Our last experiment addressed our algorithms' run-time overheads. We measured the overhead of an implementation on a Sparc 10 running SunOS 4.1.3. This implementation instruments programs at the assembly level using the techniques outlined earlier. Our implementation detects window boundaries by using an interval timer, instruments *all* Sparc instructions that reference memory, and the programs we tested were linked with instrumented versions of libraries and system-call hooks. We analyzed several executions that ran long enough to be difficult to debug without a replay tool. We ran longer executions of *mesh* and *event*; *mesh* performed 30,000 iterations over the mesh, and *event* ran two scheduling algorithms on 30,000 randomly generated graphs of about 700 nodes each. We ran a *life* program which computed 8500 generations on a 100×100 grid. To test interactive programs, we played a 20-minute game of *nethack*, and edited this section of the paper with

	mesh	*event*	*life*	*nethack*	*vi*
Uninstrumented running time (min)	16.48	40.75	8.25	≈ 20	≈ 10
Unique writes:					
running time (min)	35.46	75.01	14.53		
slowdown	2.15	1.73	1.76		
trace size (kbytes)	34,248	44,266	1,760		
trace rate (kbytes/sec)	16	10	2		
bitvector size (bytes)	16,384	16,384	4,096		
Unique-spanning$_0$ reads:					
running time (min)	112.7	321.01	23.0		
slowdown	6.83	7.87	2.78		
trace size (kbytes)	35,847	43,288	920		
trace rate (kbytes/sec)	5	2	0.67		
bitvector size (bytes)	16,384	16,384	8,192		
Unique-spanning$_\infty$ reads:					
running time (min)	113.0	319.13	21.49	≈ 20	≈ 10
slowdown	6.85	7.83	2.60	not noticeable	not noticeable
trace size (kbytes)	35,848	41,227	880	86	208
trace rate (kbytes/sec)	5	2	0.68	0.07	0.367
bitvector size (bytes)	16,384	20,480	8,192	180,224	36,864

Figure 7. Overheads of tracing algorithms.

an instrumented version of the *vi* editor. These two programs are very interactive, performing little computation between keystrokes. We measured each program's original execution time, the slowdown incurred by each tracing algorithm, the trace size and rate of trace generation, and the total size of all bitvector segments allocated during the run.

Figure 7 shows the results. The window length was chosen to be approximately five seconds of *uninstrumented* execution time. Since the slowdowns for each type of tracing are different, we chose a window length of 10 seconds for unique write tracing (since it runs about twice as slow as the uninstrumented program), and 30 seconds for unique-spanning read tracing.

Unique-write tracing incurred the least slowdown, usually a factor of two, although slowdowns are greater for programs that write to many distinct locations. Having no read hook, and having to write out the trace only at a window's end, takes less time on a Sparc 10 than writing the extra trace data that unique writes incur. Even though unique-spanning read tracing records less, the time taken by the algorithms' on-line analyses is currently greater than the cost savings of reduced disk I/O.

However, the run-time *vs* replay-time tradeoff must also be considered. In cases where the time to setup the state for a replay must be low, unique-spanning$_0$ reads have the advantage. In cases where the trace size must be minimized, unique-spanning$_\infty$ reads have the edge, although our data suggests that the trace size of unique-spanning$_0$ reads is competitive. Thus, we expect that users would currently choose between unique writes (when overhead must be minimized) and unique-spanning$_0$ reads (when replay setup time must be minimized). In addition, as machines become faster and the gap between CPU speed and disk I/O continues to widen, the slowdown of unique writes should start to approach that of the others. Eventually the run-time analysis required for detecting unique-spanning reads will be cheaper than writing the extra trace required by unique writes.

An important finding is that the rate of trace generation is acceptable (less than about 15 kilobytes/sec) even though the window length was only 10 – 30 seconds. This trace rate suggests that a gigabyte disk will suffice to trace executions of a day's length without requiring trace compression (as discussed in Section 4.1). With trace compression, executions of several day's length can be accommodated. As suggested by the curves in Figure 4, we also found that the rate of trace generation depends on the selected window length with respect to the execution's locality of reference. Choosing too small a window length for executions with large working sets can increase trace rates dramatically (for a window length of one second, *mesh*'s trace rate increases to 143 kilobytes/sec for unique writes and to 32 kilobytes/sec for unique-spanning reads). Experimentation is required to determine values that work well, although very small window lengths (under five seconds) are often impractical. The window lengths we chose above (about five seconds of uninstrumented running time) allows *any* five-second portion of the original executions to be replayed.

Our tracing algorithms perform particularly well on interactive programs. The slowdown of the instrumented versions of *nethack* and *vi* was not noticeable. The interactive nature of these programs caused traces to be generated at a low rate ($70 - 367$ bytes/sec for unique-spanning$_\infty$ reads). We expect our tracing techniques to have low enough overhead on interactive programs that they can be left on during the entire testing and debugging phase.

Figure 7 also shows that the space overhead of the bitvector segments is low. We used four kilobyte segments, which required a 128 kilobyte table of segment pointers for each bitvector (space for this table is not included in the figure). The pointer table size can be reduced by increasing the segment size, and we expect that this overhead is not significant.

7. Conclusions

Our adaptive tracing strategies make possible the trace-and-replay debugging of long-running programs that interact with their environment. We can automatically trace long-running programs with a $1.75 - 7$ times slowdown, generating less than 15 kilobytes/sec of trace (and sometimes as little as 2 kilobytes/sec). Interactive programs can be traced with trace generation on the order of 100's of bytes/sec and no noticeable slowdown. The run-time slowdowns are probably acceptable given the functionality that is being provided, especially for interactive programs. Although our implementation was carefully tuned, we see more short-term improvements possible, such as optimizing the read and write hooks more by hand, exploring the use of dedicated registers (e.g., to store the address of the bitvector segment pointer table), and the use of additional processors on a multi-processor workstation. In addition, we expect that static analysis might be able to provide a significant reduction in overhead by determining that some references need not be analyzed at all (because they will either never be traced or will always be traced). Such improvements are left to future work.

Acknowledgements

We thank David Vorbrich for discovering that the optimal tracing problem is not NP-hard and for implementing some of the simulations. Steve Reiss wrote the vmon tool used to obtain the read/write hooks for some of the simulations.

References

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford, "An Execution-Backtracking Approach to Debugging," *IEEE Software*, pp. 21-26 (May 1991).

[2] Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," *Symp. on Principles of Programming Languages*, pp. 59-70 (January 1992).

[3] Jong-Deok Choi and Janice M. Stone, "Balancing Runtime and Replay Costs in a Trace-and-Replay System," *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 26-35 Santa Cruz, CA, (May 1991).

[4] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer, "Techniques for Debugging Parallel Programs with Flowback Analysis," *ACM Trans. on Programming Languages and Systems* **13**(4) pp. 491-530 (October 1991).

[5] Stuart I. Feldman and Channing B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, (May 1988).

[6] James R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software — Practice and Experience* **20**(12) pp. 1241-1258 (December 1990).

[7] James R. Larus, "Efficient Program Tracing," *IEEE Computer* **26**(5) pp. 52-61 (May 1993).

[8] Kai Li and W. K. Fuchs, "Compiler Assisted Static Checkpoint Insertion," *Proc. of Fault Tolerant Computing Systems*, (1992).

[9] J. M. Mellor-Crummey and T. J. LeBlanc, "A Software Instruction Counter," *Proc. of the Third ASPLOS*, (April, 1989).

[10] Barton P. Miller and Jong-Deok Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 135-144 Atlanta, GA, (June 1988).

[11] James S. Plank and Kai Li, "Faster Checkpointing with N+1 Parity," *Tech Report CS-93-219*, Univ. of Tennessee, (Dec 1993).

[12] Steven P. Reiss, "Trace-Based Debugging," *AADEBUG '93*, Linkoping, Sweden, (May 1993).

[13] Robert Wahbe, Steven Lucco, and Susan L. Graham, "Practical Data Breakpoints: Design and Implementation," *SIGPLAN'93 Conference on Programming Language Design and Implementation*, pp. 1-12 Albuquerque, NM, (June 1993).

[14] Paul R. Wilson and Thomas G. Moher, "Demonic Memory for Process Histories," *Proc. of the SIGPLAN '89 PLDI Conf.*, pp. 330-343 (June 1989).