Abstract Execution in a Multi-Tasking Environment

David Mazières
Michael D. Smith

TR-31-94

November 1994

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Abstract Execution in a Multi-Tasking Environment

David Mazières Michael D. Smith

Abstract

Tracing software execution is an important part of understanding system performance. Raw CPU power has been increasing at a rate far greater than memory and I/O bandwidth, with the result that the performance of client/server and I/O-bound applications is not scaling as one might hope. Unfortunately, the behavior of these types of applications is particularly sensitive to the kinds of distortion induced by traditional tracing methods, so that current traces are either incomplete or of questionable accuracy. Abstract execution is a powerful tracing technique which was invented to speed the tracing of single processes and to store trace data more compactly. In this work, abstract execution was extended to trace multi-tasking workloads. The resulting system is more than 5 times faster than other current methods of gathering multi-tasking traces, and can therefore generate traces with far less time distortion.

1 Introduction

1.1 The Need for Multi-Tasking Address Traces

As computers continue to get faster, the gap between CPU speed and memory and I/O subsystems will continue to widen. Popular benchmarks such as the SPEC benchmark suite have tended to deemphasize the importance of this gap on overall system performance by systematically including only applications that stress raw CPU power over memory and I/O bandwidth. Of all the SPEC benchmarks, the one that seems to be scaling the least well with processor speed is *gcc*. A recent study of Ultrix performance on an R2000[NUM92] found that the *gcc* benchmark also spends much more time in system calls than any of the other benchmarks. Furthermore, the SPEC *gcc* benchmark is only the middle phase of the compiler. When the preprocessor and linker are also executed, the problem becomes even worse. Other studies across various platforms have shown that kernel performance is not scaling with processor performance at all[Ous90].

The problem is that many applications do not behave like the SPEC benchmark suite. In particular, CPU-bound programs such as those in SPEC allow the operating system to schedule processes in a way that benefits cache performance. Client/server applications, on the other hand, often have more contexts competing for the CPU and induce frequent context switches by passing messages back and forth. Moreover, many of these applications are sensitive to latency as well as throughput. In a recent study of the X11 Window system, Bradley Chen[Che94] demonstrated the serious impact that this can have on cache performance, particularly when combined with other particularities of the X Window system such as extremely large text segment sizes.

In order specifically to address the performance of these other types of applications in future operating system and hardware designs, we need both an understanding of exactly where current computers are failing to scale in speed with the raw integer performance of their CPUs, and an ability to predict the performance of memory- and I/O-bound workloads on new architectures. The most exact way of understanding performance is through tracing the execution of software: Recording the address of every single instruction that is executed, and the address of every single load or store that takes place. Such traces have been relatively easy to obtain for single processes, and have been successfully used in analyzing cache performance under SPEC-like applications. In order to understand the interaction of processes with each other and the system, however, one must acquire multi-tasking and system address traces. Several methods have been used to gather multi-tasking and system address traces successfully. However, in all cases the data have been incomplete, the act of tracing has induced distortion in the results, or both. This work shows how recent improvements in the efficiency of tracing single processes can largely improve the accuracy of multi-tasking traces.

1.2 Existing Approaches to Tracing

Existing execution tracing methods fall into two basic categories: Hardware approaches and software approaches. Hardware tracing involves physically monitoring the pins of a CPU to capture all transactions on the processor-memory bus. Programs are typically modified to emit special markers upon entry to and exit from particular regions of code. This allows tracing hardware to determine specific information about the system activity. For example, one can generate markers upon entry and exit of the kernel idle loop to determine exactly how much time is spent there. The significant advantage of hardware tracing is that it only minimally changes the system's behavior, and hence can give almost exact timing results. It does, however, severely limit the kind of information one can gather, since all analysis must be performed in real time by the CPU-monitoring hardware. Furthermore, hardware tracing scales poorly to faster and more complex processors, making it extremely difficult to use on newer CPUs.

With software tracing, one executes programs in such a way as to record memory references in main memory, where they can either be directly used in simulation or written out to some form of secondary storage. A special case of software tracing involves modifying microcode to generate trace data automatically. More generally, however, programs can be traced through simulation or single-stepped execution, or a program can actually be modified to generate its own trace information as it executes. Software methods unfortunately slow program execution down by at least an order of magnitude, and often several. Such a slowdown can change the order of events and affect the validity of gathered traces. However, with the exception of microcode modification, software tracing scales well to newer and faster processors. Software traces have already been used for cache analysis[Aga89, BKW90] with great success. By optimizing the tracing process to reduce slowdown, we can obtain traces accurate enough to analyze network-bound applications or simulate radically different processor architectures. The remainder of this section outlines current tracing methods.

1.2.1 Hardware Tracing

A typical hardware tracing tool is Monster[NUM92]—a Tektronix DAS 9200 logic analyzer connected directly to the pins of the MIPS R2000 microprocessor in a DECstation 3100. The Ultrix kernel is modified to emit special markers at the entry and exit of certain regions of code. The state machine in the logic analyzer is then programmed to keep an exact count of events one is interested in studying. The markers are used to restrict analysis to particular regions of code, or simply to record how much time is spent in a given area such as the kernel idle loop. This enables one to track down system performance bottlenecks with great precision.

There are, however, several drawbacks to Monster. It functions well as a counter of events, but generating a full address trace for later simulation is much more difficult. The logic analyzer only contains 512K of RAM, and though it can be triggered to start and stop tracing at specific points of execution, it cannot generate a full address trace over a long period of time. True, one can build logic analyzers with arbitrary amounts of memory, but tens of megabytes of raw data would be generated every second. The act of writing the information to some form of secondary storage would require far longer than actually acquiring the data, and this would leave significant gaps in the trace. Thus, while Monster can give an accurate picture of performance on a DECstation 3100, it is unsuitable for simulating the behavior of systems with different cache or TLB sizes and structures.

It is also difficult and expensive to build a hardware tracing tool like Monster that can sample at the speeds of the newest processors. The R2000 processor analyzed by Monster runs at 16 MHz and has no internal cache. Current processors running at close to ten times that speed will require significantly faster hardware to do the monitoring. Furthermore, as processor speeds continue to increase, CPU to memory buses may not be able to function properly under the additional capacitative loading of a logic analyzer. Already, new high-speed memory systems such as Rambus[Ram] operate in a terminated environment and rely on special packaging to shorten bus lengths and control capacitance.

A further difficulty ensues from the increasing amount of hidden state in microprocessors. It is common, for instance, for the external interface of a processor to function at a fraction of the internal speed. Additionally, determining which instructions are meaningfully executed and when the pipeline is flushed becomes difficult in multiple-issue processors with delayed load stalls and complicated store buffers. Moreover, as multi-ported on-chip caches with other accompanying structures such as branch-target buffers become integral parts of processor design philosophies, the problem of monitoring activity between the processor and cache will become almost insurmountable.

1.2.2 Software Tracing

Software approaches to tracing sacrifice the unobtrusiveness of hardware tracing for the ability to scale with processor speed and complexity. The most basic form of software tracing consists of recording all interesting events that occur as one single steps a program through use of the Unix system call `ptrace`.[1] This approach is easy to implement and scales well to faster processors, but introducing system call and context switch overhead for every traced instruction executed will slow the traced process down by a thousand times or more. Such a slowdown makes a `ptrace`-based approach unsuitable for tracing anything but the simplest of programs.

An approach similar to `ptrace`-based methods is that of simulating software execution to obtain trace data. One of the most advanced software simulators is *Shade*[CK]. *Shade* allows one to trace only a particular subset of instructions such as all loads and stores. Highly optimized code is constructed at run time to execute blocks of the simulated instructions on the processor in real time. Whenever a traced instruction appears, *Shade* invokes a user function to perform whatever analysis is desired by the user. Even with empty user analysis functions, however, a load and store traced simulation of a typical integer benchmark will run 15 times slower than the original program.

In an attempt to combine the benefits of hardware tracing with the flexibility and completeness of software tracing, Agarwal[Aga89] modified the microcode of a VAX 8200 to generate a full address trace of all instructions executed, a scheme he called Address Tracing Using Microcode (ATUM). In ATUM, the modified microcode reserves a portion of main memory on bootup, and makes that region inaccessible to the operating system. The `XFC` "microcode extension" instruction can be used to turn tracing on or off, and to read words from reserved memory. Typically, a process enables tracing and sleeps until the buffer fills. When the buffer fills, tracing is automatically shut off and the sleeping process awakens to read back the trace data and write it to secondary storage. Once the buffer has been emptied, tracing is resumed.

ATUM has been used very successfully for detailed cache analysis. It has a significant advantage over other software tracing methods in that tracing the operating system is no more difficult than tracing user processes. ATUM does, however, induce a ten-times slowdown, which is compensated for by slowing the clock interrupt. Another disadvantage of ATUM is the distortion caused by processing the buffer. When the buffer fills up and tracing is turned off, the system continues to run as usual, making exact traces of more than half a million references impossible. To analyze the system over long periods of time, then, one must simulate long traces by stitching together shorter ones. This, in turn, requires compensating for the gaps while analyzing the data. A more fundamental problem with ATUM is the fact that it cannot be implemented on RISC processors with hardwired control.

In order to bring the slowdown of software tracing to a usable level without relying on microcode, then, most software tracing programs actually embed tracing code in the executable image of the program being traced. This process, known as "instrumenting" an executable, can occur at compile-time, link-time, or after link-time. One of the most well-known such software tracing tools is Earl Killian's tool *pixie*[Smi91]. *Pixie* modifies preexisting MIPS executables to emit all the information necessary to regenerate a full instruction and data address trace. When *pixie* instruments an executable for instruction and data tracing, it adds instrumentation code at the beginning of every basic block,[2] and at each load and store instruction. The markers are eventually written to file descriptor 19 of the process being traced, where they can easily be decoded into an address trace and run through a simulator.

A *pixie*-instrumented executable will generate a full instruction and data address trace of a program in approximately 15 times the original, untraced execution time of the program. With this tolerable slowdown, *pixie* has proved an invaluable tool for analyzing the performance CPU-bound programs such as the SPEC benchmark suite. But *pixie* is not capable of meaningfully tracing all programs. In particular, certain assumptions about a program being "well-behaved" must be made when rewriting an executable. Often

[1] The details of `ptrace` are described in the manual pages of BSD 4.4, as well as most other Unix operating systems.
[2] A basic block is a group of instructions which is always executed from start to finish.

hand-written assembly code—and occasionally compiler-generated code—will not obey these assumptions. *Pixie*, for instance, since it changes the addresses of all functions by adding instrumentation code, is unable to translate the address of a function that is passed to the `sigvec` system call. As a result, *pixie* cannot trace programs with signal handlers.

Specific inconveniences such as the inability to handle signals could be fixed up on a case by case basis. There are, however, some fundamental limitations to tracing with a tool like *pixie*, which become more immediate concerns if one attempts to use the same approach to trace a multi-tasking workload. Though the speedup of *pixie* over `ptrace`-based methods makes it possible to gather useful traces in a reasonable amount of time, the remaining slowdown is enough to make most asynchronous events appear to occur instantaneously. This, combined with any extra page-faults incurred as a result of a text segment dilated with instrumentation code, can affect the behavior of the scheduler and distort multi-tasking results.

Nonetheless, *pixie*-like tools have successfully been expanded to generate exact traces of multi-tasking workloads. Borg, Kessler, and Wall[BKW90] developed such a tool to trace Titan workloads at the DEC WRL. Their system uses link-time instrumentation to insert branches to tracing code at every load, store, and basic block entrance. This enables them to capture all loads, stores, and instruction fetches, but does not record the location of loads and stores within their basic blocks. For efficiency, they reserve 5 of the 64 general-purpose registers in the Titan for use by instrumentation code. All trace information is written to a global buffer which is mapped into the top of every process's virtual address space. At each entry to or exit from the kernel, a marker containing the current process id and whether the switch is into or out of the kernel is written to the buffer. When the buffer overflows, all traced processes block and a special high-priority process is scheduled to compress and store the buffer contents, or to use it immediately in simulation.

The execution time of instrumented executables in the Titan tracing system is 8–12 times longer than normal. To compensate for this time-dilation and to increase the applicability of the simulations to faster, newer RISC processors, the process switch time is increased by a factor of 16. Unfortunately, cache simulations tend to consume data 10 times more slowly than this system can create it. Therefore, with a 32 megabyte buffer filling up approximately every second, this system is forced to suspend execution of programs for long periods of time. Despite this slowdown, the Titan work was able to make excellent predictions about the behavior of very large caches with various degrees of associativity. The ideas developed during the Titan project form the basis of the approach described in this document.

Bradley Chen[Che93, Che94] at CMU later extended the Titan tracing method to trace the MIPS processor using a modified version of the tracing tool *epoxie*[Wal92]. *Epoxie* modifies executables prior to linking them, so as to reduce the size of the rewritten executable. One typically combines all object modules into one big object module, using the command `ld -r`, runs *epoxie* on the resulting object file, and finally turns the output of *epoxie* into an `a.out` format file by invoking *ld* a second time. Thus, while tracing tools that rewrite `a.out` executables need to translate register jumps at runtime from large translation tables, *epoxie* manages to have the linker perform translations, and therefore produces much smaller executables.

Unfortunately, the MIPS architecture contains only 32 registers: One is hardwired to zero, two are reserved by the kernel for fast TLB miss handling, and the other 29 may all be used by the compiler.[3] Thus, where the Titan tracing system could simply reserve five registers for use by the instrumentation code, *epoxie* must actually perform "register stealing," which means saving and restoring "shadow registers" from memory wherever the stolen registers are used in the original object file. This vastly complicates mapping a single global buffer into every address space, as the current location cannot be shared between processes in a register. As a result, each traced process maintains its own buffer in Chen's system. At every entry into the kernel, the contents of the current process's trace buffer is copied by the kernel into a system trace buffer, and the trace is consumed or stored whenever the system trace buffer overflows.

Chen's system is also able to trace both the Ultrix and Mach kernels. In each case, slightly under 1,000 lines of `locore.s` were instrumented by hand, while the rest of the kernel was instrumented automatically by the modified *epoxie*. Since instrumented executables run approximately 15 times slower, the clock is also slowed down by a factor of 15. TLB misses do occur more frequently because of larger text segments, but instead of relying on traces of the user[4] TLB miss handler to study TLB misses, the TLB can be simulated

[3] In reality, one MIPS register, `$1`, is reserved for the assembler to expand compiler-generated pseudo-instructions.

[4] Most of the kernel resides in unmapped memory for MIPS operating systems. Hence the kernel does not cause many TLB misses.

from the user instruction address traces which correspond to locations in the uninstrumented executable.

Of course, I/O requests still appear to complete 15 times more quickly than they would during untraced execution. Tracing therefore affects scheduler policy despite slowing down the system clock. Scheduler policy, in turn, affects the performance of CPU-bound workloads, where processes scheduled for longer periods of time experience better cache performance because of temporal locality. These CPU-bound processes, however, are precisely those which can already be analyzed with a tool such as *pixie*. Chen therefore concentrated on analysis of applications where client/server behavior dictates process scheduling.

1.3 Our Approach

The passivity of hardware tracing approaches is quite desirable. Unfortunately, as we have seen, hardware tracing is not general enough to simulate arbitrary architectures, and will no longer be feasible on newer processors. Software tracing, on the other hand, though more difficult, has already proven to be both possible and useful. The most serious difficulty it presents is time dilation. Not only will a 10–15 times slowdown affect process scheduling, it will give I/O the appearance of being an order of magnitude faster than it actually is.

Now that software tracing of multi-tasking and system workloads has been shown to be possible, we must increase its usefulness by finding ways of minimizing the drawbacks, time-dilation in particular. This will allow us to gather useful traces of I/O-bound workloads, in particular network applications. The work presented here shows how tracing overhead can be reduced more than five times through the use of a technique called Abstract Execution[Lar90], in which much of the overhead of producing a full address trace can be postponed to analysis time. The system described in this paper is based on the tool *qpt*, which uses Abstract Execution to generate very efficient address traces of a single process.

Qpt was modified to work in a multi-tasking environment, and support for the tracing was added to the 4.4 BSD kernel. Extending *qpt* to trace multiple processes had no negative impact on performance. In fact, certain optimizations were added to *qpt* that would not have been possible without kernel support. For example, costly bounds checking on the buffer is now performed automatically by the VM system. The current implementation only traces user level processes; however, it has been designed with the objective of eventually being able to trace the kernel.

Section 2 describes abstract execution, a powerful optimization to traditional methods of tracing a single process. Section 3 explains how we extended abstract execution to trace multiple processes. Section 4 describes the results of our approach, concentrating specifically on performance of the first implementation. Section 5 discusses further optimizations that can be made to the current system, and how it can be extended to trace the kernel. Finally, we end with a brief conclusion in section 6.

2 Abstract Execution

2.1 Motivation for Abstract Execution

`Ptrace`-based tracing methods suffer from too much information being made available about the traced process: For each instruction traced, the program under study is completely frozen, and its entire state is made available to the parent process, which must look at the instruction, decide if the instruction is interesting, and if so what registers in the child process also need to be looked at. With *pixie*, on the other hand, the program is modified to do its own tracing, and output only the information considered interesting—an address trace in our case.[5] In order to do this, *pixie* must add code for every load, store, and basic block entry. Turning a traced program's *pixie* output into an address trace is completely trivial, but program execution is slowed down significantly to produce such full output.

The best solution for tracing, then, is to take the advantage of *pixie* one step further—to record far less than a full address trace at execution time. The program under study should only emit markers for a minimal subset of interesting events at run time. The rest can be deduced at a later point as one analyzes the trace data. That way time dilation on an instrumented executable can be minimized, but a full address trace can still be generated without nearly the overhead of simulating the program under study. The idea

[5] *Pixie* can also profile programs by simply counting the number of times each basic block is executed.

```
while (P) {
  if (Q)
     A;
  else
     B;
  C;
}
```

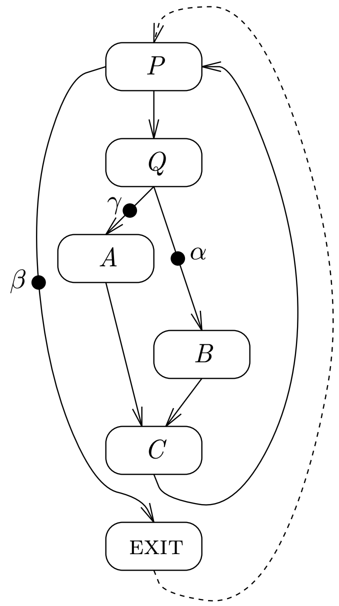


Figure 1: Simple procedure and corresponding CFG.

of instrumenting an executable but deferring parts of trace generation to a later time is the key to Abstract Execution[Lar90]. Section 2.2 explains which subsets of events it is sufficient to record at runtime, and gives a method for choosing such a subset with a low execution cost. Section 2.3 describes the specifics of implementing abstract execution on the SPARC.

2.2 Overview of Abstract Execution

Let us first consider the problem of minimizing tracing overhead while gathering a full instruction trace of a single procedure; later we will see that tracing multiple procedures and generating data traces are extensions to the same basic approach. Figure 1 shows a very simple procedure and its associated control flow graph: Each node in the graph corresponds to a basic block of the program, and each directed edge to a possible transfer of control flow. Let us call the set of vertices in this graph V and the set of edges E. We wish to insert instrumentation along edges and in basic blocks of the program so that every possible execution path from P to EXIT leaves a different trace. In addition, we would like to minimize the number of times that instrumentation code needs to be executed during a typical execution of the procedure. Given a mixed set of edges and vertices which have been instrumented, one can, for each instrumented vertex, instrument instead the set of all its outgoing edges. Since, by the definition of basic block, a vertex is traversed exactly as many times as the sum of all its outgoing edges, the exchange cannot increase the number of times instrumentation code is invoked in any execution of the program. Figure 2, however, is an example showing that instrumenting vertices can be more expensive than instrumenting edges. If, for instance, one must instrument only vertices, instrumentation code will have to be added to A and B. Execution path $P \to A \to B \to$ EXIT will execute two pieces of instrumentation code. If, on the other hand, the two dotted edges are instrumented, only one piece of instrumentation code needs to be executed per invocation of the procedure. For this reason, we always choose to instrument edges rather than vertices.

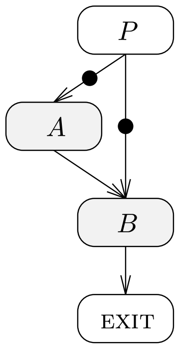


Figure 2: Edge and block trace.

2.2.1 Terminology

Now we must determine a way of choosing which subset of the edges in a program to instrument. The explanation given here is designed to explain the motivation for the edge-picking algorithm used with Abstract Execution. A more formal treatment may be found in [BL91]. We first need to define some terminology:

- Any vertex with two or more outgoing edges is known as a *predicate*. In Figure 1, P and Q are both *predicates*.
- A marker emitted by an instrumented edge is known as a *witness*. If, for instance, the black dots on edges represent instrumentation code in Figure 1, we say that edge $Q \to A$ writes witness γ.
- We say there exists a *witness-free* path between two vertices a and b of a CFG if there exists a set of zero or more vertices x_i such that there exist edges $a \to x_1 \to x_2 \to \cdots \to b$ which do not write witnesses. If we represent instrumented edges with dots as in Figure 1, this means that there exists a directed path from a to b which does not cross any black dots.
- Given a CFG, the *witness set* of a predicate p with respect to an edge $p \to q$, written $witness(p, q)$, is defined as follows:

  If $p \to q$ writes witness τ, then $witness(p, q) = \{\tau\}$.

  If $p \to q$ does not write a witness, then $witness(p, q)$ contains any witness υ such that there exists a *witness-free* path from q to some vertex x, and an edge $x \to y$ which writes υ.

  In addition, if $p \to q$ does not write a witness and there exists a *witness-free* path from q to EXIT, $witness(p, q)$ also contains the special witness EOF.

  Put more simply, a witness is in $witness(p, q)$ if it can be the next witness recorded when edge $p \to q$ is executed. In the case of the example CFG, $witness(P, Q) = \{\alpha, \gamma\}$.

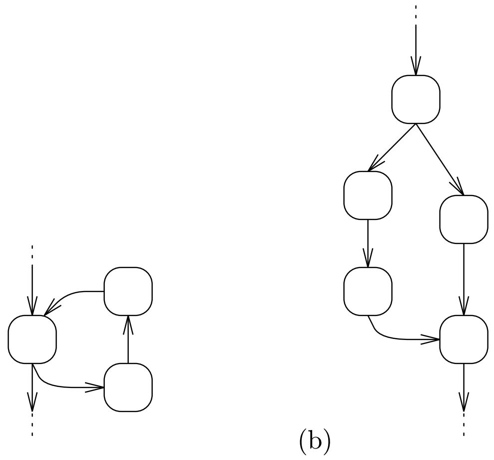
2.2.2 Solving the Tracing Problem

The problem of being able to regenerate an instruction address trace from a set of witnesses is equivalent to determining exactly which outgoing edge was executed from each predicate. Given a predicate p and outgoing edges $p \to q_1$, $p \to q_2$, ... , $p \to q_n$, we will be able to determine which outgoing edge was taken from the next witness if and only if $witness(p, q_i) \cap witness(p, q_j) = \emptyset$ when $i \neq j$. In other words, if some witness τ can potentially be the first witness recorded when either of two distinct edges $p \to q_i$ and $p \to q_j$ is traversed, then there will be no way of disambiguating the two edges when τ is read back from the witness file after executing the instrumented program.

If a CFG has the set of vertices V and the set of edges E, then a set of edges $epl \subseteq E$ solves the *tracing problem* when one can instrument only the edges in epl and still have disjunct *witness-sets* for the outgoing edges of every predicate in V. There are two cases which clearly do not satisfy the *tracing problem*, illustrated in Figure 3 on the next page: If $E - epl$ contains a directed cycle, in other words there is a witness-free path containing a program loop, there will be no way of knowing how many times the loop is executed. In our example of Figure 1 on the preceding page, suppose that edge $Q \to B$ were not instrumented. The path $P \to Q \to B \to C \to P$ is now witness-free. Thus β is now in $witness(P, Q)$ as well as $witness(P, \text{EXIT})$. If at vertex P we read β from the witness file, we will have no way of knowing if the next vertex is Q or EXIT. Another case which does not satisfy the *tracing problem* is when $E - epl$ contains a diamond. A set of edges contains a diamond if it contains loop-free paths $p \to a \to \cdots \to x$ and $p \to b \to \cdots \to x$ which are disjoint except for p and x. It is clear that $witness(p, a) \cap witness(p, b)$ will contain the first witness of any path between x and EXIT. (If no such witnesses exists, then both witness-sets will contain EOF.)

Conversely, if a subset of edges epl does not solve the tracing problem, then $E - epl$ must contain a diamond or loop. To see this, suppose that $\alpha \in witness(p, a) \cap witness(p, b)$ where $a \neq b$, and that α is generated by edge $x \to y$. There must exist two witness-free paths from p to x, one of which starts with edge $p \to a$, and the other with $p \to b$. Let z be the first vertex these two paths have in common (z may be the same as x). There are two witness-free paths $p \to a \to \cdots \to x$ and $p \to b \to \cdots \to x$ which are disjoint. Either one of them contains a loop, or the pair of them forms a diamond. Therefore, the sets of edges which solve the tracing problem are precisely those whose complements do not contain directed cycles or diamonds.

A *weighting* of a CFG is the assignment of a non-negative value to each edge of a graph in such a way that the sum of the weights of incoming edges to a node is always equal to the sum of outgoing edges. If we add an imaginary edge from the EXIT block to the first block of a procedure (represented by the dashed



(a) (b)

Figure 3: Impossible *witness-free* paths: (a) Loop. (b) Diamond.

arrow in Figure 1 on page 6), then the execution counts of the edges in a procedure form a weighting (call it W_{EX}) of the procedure's CFG, provided that the imaginary edge is assigned the number of times the procedure has executed. The problem of executing instrumentation code the least number of times, then, is the problem of finding the set of edges *epl* which solves the tracing problem with the lowest possible sum of weights in W_{EX}.

Of course, W_{EX} is generally not known before the program has been traced or profiled, and will generally change from execution to execution if the input events of the program are not identical. However, some sets of edges are always better than others. For example, in Figure 1, the weight of $P \to Q$ is equal to the sum of the weights of $Q \to A$ and $Q \to B$. Thus, it is always better to instrument $Q \to A$ and $Q \to B$ rather than, say, $P \to Q$ and $Q \to A$, even though one can solve the tracing problem with either pair. Thus, one can benefit from finding a solution to the tracing problem which minimizes the weights of the instrumented edges with respect to some weighting, even if that weighting is not the actual execution weighting. One way to obtain such a weighting is to profile or trace the program once with some non-optimal edge placement, and to use the results in instrumenting the executable a second time. As we will see, however, when multiple procedures are traced, there are further restrictions on the placement of instrumentation code which make a two-step instrumentation process not worthwhile. The program *qpt* used for this multi-tasking tracing work simply calculates a weighting on the assumption that all loops iterate 10 times, and that **if** conditionals are true half of the time.

Minimizing the total weight of *epl* is the same as maximizing $E - epl$. Unfortunately, finding the maximally-weighted set of edges that does not contain a directed cycle or diamond in any given CFG is not always an easy problem. A slightly more restrictive requirement is that $E - epl$ contain no directed or undirected cycles whatsoever. A set of edges that reach every vertex of a graph without cycles is simply a spanning tree. A maximum spanning of a CFG can easily be found using Kruskal's algorithm:[6] One starts with an initially empty set of edges and keeps adding the heaviest edge that does not form a loop until finally one can add no more edges without creating a loop. Larus and Ball[BL91] have shown that for a large class of CFGs, choosing a maximum spanning tree for $E - epl$ does in fact produce optimal results. For the other types of CFG that exist, they argue that the spanning tree approach is still near-optimal. This approach is therefore used to determine *epl* in their implementations of Abstract Execution.

There is still one slight qualification that needs to be made to the above method. The edge from EXIT to the first block of the procedure is imaginary, and therefore cannot be instrumented. If the spanning tree does not contain this edge, however, there will be paths through the procedure which do not write any witnesses to the trace file. Consider, for instance, what would occur in Figure 1 if β were to lie on the dotted arrow rather than on edge $P \to$ EXIT: We would have no way of knowing how many times basic block P executed,

[6] A more detailed description of Kruskal's algorithm and a proof of its optimality may be found in [LD91]. The explanation is based on finding a minimum spanning tree, but the logic applies equally well to finding a maximum spanning tree.

because we would not know how many times the procedure executed. We must therefore begin Kruskal's algorithm with the imaginary edge already in the spanning tree.

2.2.3 Tracing multiple procedures

First, let us consider the case of a program in which control transfers between procedures only through ordinary call instructions. Suppose that vertex Q of Figure 1 contains a call to another procedure—let us call such a vertex a *call vertex*. The called procedure will also write witnesses to the trace file, including possibly a witness with the same value as β. When regenerating the basic block trace, then, if we see the witness β at vertex P, we will not know whether to follow the edge to EXIT or proceed to Q and interpret β as the first witness of the called procedure.

One could possibly avoid this problem by assigning all procedures different witness numbers, though this would severely impact both trace size (since 8-bit witnesses would no longer be usable for small functions), and trace compressibility. However, suppose the procedure called by Q calls its parent procedure in the first basic block, and that this time edge $P \rightarrow$ EXIT is taken—This possibility still prevents us from regenerating a trace. Thus, to make such a scheme work, all procedures would have to emit a witness upon entry to prevent the possibility of mutual recursion from introducing ambiguities. Furthermore, the added complexity of writing data trace information to the same buffer will make a unique witness number approach even less practical.

To reconstruct a multi-procedure trace, then, we must always know which procedure wrote a byte at the point where we read that byte from the trace file. Ambiguity will arise whenever there exists a witness-free path from a predicate to a call vertex. We must therefore add a witness along each path from a predicate to a call vertex, to block the regeneration program from seeing the child procedure's witnesses when it might also see a witness from the current procedure. Such a witness is called a *blocking witness*.

However, if blocking witnesses are not added carefully, more edges may end up being instrumented than necessary. In our example, when we add a blocking witness to edge $P \rightarrow Q$, call it δ, we no longer need witnesses on both $Q \rightarrow A$ and $Q \rightarrow B$. If, for instance, we eliminate witness γ along $Q \rightarrow A$, $\mathit{witness}(Q, B) = \{\alpha\}$ and $\mathit{witness}(Q, A) = \{\beta, \delta\}$, so that the resulting set of instrumented edges still solves the tracing problem. In general, therefore, it is better to place blocking witnesses before calculating the maximum weight spanning tree, as this will prevent redundant witnesses.

A second source of redundant witnesses can occur with multiple call vertices along a predicate-free path. Suppose, for instance, that vertices A and C of Figure 1 are call vertices. If one inserted a blocking witness for C along edge $A \rightarrow C$ before handling A, then one would still need to instrument edge $Q \rightarrow A$ to block the procedure call in A from predicate Q. If on the other hand, $Q \rightarrow A$ had been instrumented first, then there would be no need to instrument $A \rightarrow C$. Thus, as one proceeds through the list of call vertices to insert appropriate blocking witnesses, they should always be inserted on the outgoing edge of a predicate. Stated formally, for each call vertex a and predicate p, if there is exists a set of zero or more nodes x_i which are not predicates and a path $p \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow a$, we must insert a blocking witness on edge $p \rightarrow x_1$, or $p \rightarrow a$ in the case of no x_i's.

Many programs, however, do not jump from one procedure to another exclusively through call instructions. The C function `longjmp`, for instance, loads an address from a `jmp_buf` into a register and performs an indirect jump to the register. The solution is to record the addresses of the `jmp_buf`s when they are written in `setjmp` and jumped to in `longjmp`. The regeneration program can then map each of the traced program's `jmp_buf`s to one of its own, and execute a `setjmp` or `longjmp` wherever the traced process did. Instrumenting code to record the argument of one of these functions is essentially the same problem as recording the address values of loads or stores. The same algorithm can be used for both, and is described in the next section.

2.2.4 Gathering Data Traces under Abstract Execution

Once we have a way of generating an exact instruction address trace, it becomes much easier to deduce the full set of memory data references. Any procedure with loads and stores will have a particular set of registers which determine the addresses of those loads and stores.[7] An instruction which does not modify a register

[7] As an optimization, however, certain locations on the stack can be treated as registers in the method described here.

```
char scratch[32];
void zero (char *, int);

int main ()
{
  zero (scratch, sizeof(scratch) - 1);
  return (0);
}

void zero (char *p, int i)
{
  while (i>=0) {
    p[i--] = 0;
  }
}
```

Figure 4: listing of `simple.c`

in that set is considered an *easy* instruction—it plays no role in the data trace and so can be ignored. An instruction which modifies an important register in a predictable way, such as an increment of a pointer, is considered a *difficult* instruction. When difficult instructions are encountered, however, no additional instrumentation code is required. Instead, the difficult instruction can be executed by the regeneration program. *Impossible* instructions are loads of an important register from memory, or return values from functions. Only impossible instructions require extra instrumentation code.

Let I be a memory instruction in procedure P, and R be the set of registers which determine the data address of I. (For a typical RISC architecture, R would only contain one register.) Now let $\mathcal{D}(I)$ be the set of all instructions which define one of the registers in R—in other words, the set of all instructions which might last write a register of R before execution of I. Now let $\mathcal{D}^*(I)$ be defined recursively as the set of all instructions J in P such that either $J \in \mathcal{D}(I)$, or there exists an ALU (in other words difficult, rather than impossible) instruction K such that $K \in \mathcal{D}^*(I)$ and $J \in \mathcal{D}^*(K)$. $\mathcal{D}^*(I)$ is called a *program slice* with respect to I.

In order to generate a data trace, then, we must compute the program slice with respect to each load and store instruction. The impossible (non-ALU) instructions in the slice must be instrumented. The hard instructions must be emulated at regeneration time, and instrumentation code must be added at the beginning of the procedure to record any source registers of hard instructions which are live on entry. While calculating program slices, it is trivial additionally to flag as live on entry the argument register of a procedure named `setjmp` or `longjmp`, if that is required as described in the previous section.

2.3 *Qpt* on the SPARC

2.3.1 Background

James Larus[Lar90], who coined the term abstract execution, based two tracing tools on the technique: *AE*, and *qpt*. *AE*, the earlier of the two, was a modification to the GNU C compiler *gcc* that instrumented code at compile time. As a side affect of compiling, *AE* also generated a "program schema," which described the structure of the program being compiled. A second program named *aec* would then compile the schema into C code capable of turning the trace data emitted by the compiled program into a full address trace. *Qpt*, his second generation tool, has been improved to modify executables directly. Given an executable file `a.out`, *qpt* generates an instrumented executable `a.out.qpt` and one or more C files containing code capable of turning the output of `a.out.qpt` into a full address trace.

Let us look at a simple example and see how, on the SPARC, we can trace the program `simple.c` listed in Figure 4. The result of compiling this program with `gcc -O` is given in Figure 5 on the next page.[8] Before

[8] *Gcc* version 2.5.8.

```
00002290 <_main>    save  %sp, -112, %sp              ! Block 0
00002294 <_main+4>  call  000023a4 <___main>
00002298 <_main+8>  clr  %i0
0000229c <_main+c>  sethi  %hi(0x6400), %o0           ! 000065e0
000022a0 <_main+10> or  0x1e0, %o0, %o0               !  <_scratch>
000022a4 <_main+14> call  000022b4 <_zero>
000022a8 <_main+18> mov  0x1f, %o1
000022ac <_main+1c> ret
000022b0 <_main+20> restore

000022b4 <_zero>    save  %sp, -112, %sp              ! Block 0
000022b8 <_zero+4>  cmp  %i1, 0
000022bc <_zero+8>  bl  000022d4 <_zero+20>
000022c0 <_zero+c>  nop
000022c4 <_zero+10> clrb  [ %i0 + %i1 ]               ! Block 1
000022c8 <_zero+14> addcc  -1, %i1, %i1               ! Block 2
000022cc <_zero+18> bpos,a   000022c8 <_zero+14>
000022d0 <_zero+1c> clrb  [ %i0 + %i1 ]
000022d4 <_zero+20> ret                               ! Block 3
000022d8 <_zero+24> restore
```

Figure 5: `simple.c` compiled with `gcc -O`. The "!" character begins comments.

proceeding, however, a few things must be noted about the peculiarities of the SPARC architecture and assembly language. First, unlike other platforms, the destination specifier is always last in SPARC assembly language. Second, only the first 8 registers (named `%g0`...`%g7`) are global to all functions. However, `%g0` is hardwired to zero, and `%g5`...`%g7` are reserved for future use and never touched by compiler-generated code[SPA, Appendix D]—which means instrumentation code can always make use of them. The remaining 24 registers, named `%i0`...`%i7`, `%l0`...`%l7`, and `%o0`...`%o7` take part in the register window scheme.

Each function starts off with a `save` instruction. This rotates the register windows, making register `%o`x into register `%i`x, and generating a new set of `%l` and `%o` registers.[9] The `save` instruction also acts like an add instruction, with its source in the old register window and its destination in the new one. Since the stack pointer is `%o6`, this allows a new call frame to be allocated at the same time as the register windows are being rotated, and allows the stack pointer to be restored automatically when a function executes a `restore` instruction before returning. The first six arguments to functions are passed in the `%o` registers (which then become `%i` registers in the called function), and the return address is stored in `%o7`.

There is one other odd instruction to note in Figure 5, and that is the call to function `__main` that has been inserted by gcc yet does not appear in the C source code. This function invokes constructors of any global C++ objects when one has linked with C++ modules. In the case of C programs, however, `__main` returns immediately and the call to it can be ignored.

2.3.2 Instruction and Data Tracing

If we run *qpt* on `simple`, we obtain an instrumented executable called `simple.qpt`, and a C program called `simple_sma000.c` capable of generating an address trace for `simple`. Let us examine the way *qpt* instruments this program to generate a full address trace. First, let us determine the instrumentation code we need to add in order to be able to regenerate a full instruction trace. Since `start`, the function that calls `main`, has presumably been instrumented correctly according to the method described in the previous section, there is a blocking witness before the call to `main`. We therefore already know that we are entering `main` whenever we enter it. Since `main` contains no conditional branches (it is one large basic block), we need add no instrumentation code to it to obtain a full instruction address trace.

[9] If there are no more free register windows, this instruction causes a trap to the kernel where register windows are automatically written to the process's stack, thus freeing up more register window space for further calls.

Function `zero`, on the other hand, contains conditional branches. The control flow graph of `zero` is depicted in Figure 6, along with the edge and vertex numbers assigned to the function by *qpt*. Let us find a maximum weight spanning tree of edges not to instrument in function `zero`. We start by adding the imaginary edge 3 → 0, and make that the first edge in our spanning tree. Next, we must place blocking witnesses between every predicate and call or exit vertex. In this example, this means we are forced to instrument edges 0 → 3 and 2 → 3 (which are confusingly numbered 2 and 1 by *qpt*, respectively). Finally, we must repeatedly add to our spanning tree the most expensive edge that will not cause a loop and that does not already bear a blocking witness. In this example, edge 2 → 2 (or 0, as *qpt* numbers it) is the most expensive edge, if we estimate that loops iterate 10 times. However, that edge will form a loop if added to the spanning tree. Therefore, we must add edge 0 → 1 or 1 → 2 to the spanning tree. Since these both must have the same weight, either one will be added as the second edge of the tree, and the other will become the third edge. This leaves edges 0 → 3, 2 → 3, and 2 → 2 to be instrumented.

The function `zero` also contains store instructions in blocks 1 and 2 (namely the `clrb` instructions—this opcode is an abbreviation for a byte store of register `%g0`). Let us first compute the slice with respect to the store in block 1. Working backwards from the instruction, we see that no previous instruction in block 1 affects the values of registers `%i0` or `%i1` (this is trivial, as `clrb` is the only instruction in block 1). We then check all predecessor blocks, in this case only block 0. We proceed backwards through block 0 looking for instructions which define `%i0` or `%i1`, but before finding any, we land at the first address of the function. We therefore add `%i0` and `%i1` to the list of registers which are used on entry. Next, we must compute the slice with respect to the `clrb` in block 2. Proceeding backwards, we find an `addcc` instruction which defines `%i1` in terms of itself. We therefore flag this instruction as a difficult instruction, and continue backwards looking for instructions which define `%i0` and `%i1`. As we go through predecessor blocks, we must mark any blocks we fully explore so as to help performance and avoid any infinite recursion. In this case, we will explore blocks 1 and 0, and then explore block 2 again (since 2 is a predecessor of itself), but proceed no further.

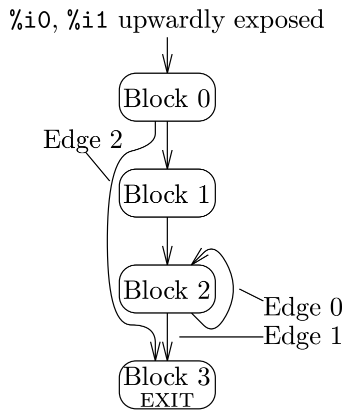


Figure 6: CFG for function `zero`

2.3.3 Rewriting an Executable

It is desirable to have the data segment at the same address in instrumented and uninstrumented versions of a program, so as to avoid having to translate all data references when rewriting an executable. However, the data segment of a program immediately follows the text segment in SPARC executables. This prevents us from enlarging the text segment of a program without changing the address of the data segment. To avoid shifting the data segment, then, *qpt* places the new text segment immediately after the data segment. To preserve the location of the bss segment,[10] a *qpt*-instrumented executable always begins execution by copying the program text to the end of the new bss segment, and zeroing out the location of the old bss segment. Any read-only data in the old text segment also remains in its old location, unmodified by the instrumentation process. The values of addresses that contained program text in the old text segment are replaced by the new addresses of equivalent instructions. This allows for text address translations in C `case` statements and functions such as `sigvec`.

A partial disassembly of `simple.qpt` is shown in Figure 7 on the next page. For readability, instrumentation code has been replaced by a description of the function. In all cases, however, a pointer to the current location in the buffer is kept in `%g6`. The value to be recorded is simply moved to register `%g7`, and stored to the buffer, after which `%g6` is incremented by the appropriate amount. `Check_buffer(`n`)` checks to see if there are n bytes available in the trace buffer, and branches off to flush the buffer if not. `Record_result(`r`)` writes the value of register r to the trace buffer. This takes six instructions because the

[10]The bss segment contains uninitialized or zero-initialized data which need not be stored in the program executable image. It always immediately follows the data segment in memory.

```
000086e4 <_zero>    check_buffer (8)
00008700 <_zero+1c> record_result (%o0)
00008724 <_zero+40> record_result (%o1)
00008748 <_zero+64> check_buffer (1)
00008764 <_zero+80> save  %sp, -112, %sp              ! Block 0
00008768 <_zero+84> cmp  %i1, 0
0000876c <_zero+88> bl  000087c0 <_zero+dc>
00008770 <_zero+8c> nop
00008774 <_zero+90> clrb  [ %i0 + %i1 ]               ! Block 1
00008778 <_zero+94> b,a   00008788 <_zero+a4>
0000877c <_zero+98> record_witness (0)                ! Edge 0
00008788 <_zero+a4> check_buffer (2)
000087a4 <_zero+c0> addcc  -1, %i1, %i1               ! Block 2
000087a8 <_zero+c4> bpos,a   0000877c <_zero+98>
000087ac <_zero+c8> clrb  [ %i0 + %i1 ]
000087b0 <_zero+cc> record_witness (1)                ! Edge 1
000087b8 <_zero+d4> b  000087cc <_zero+e8>
000087c0 <_zero+dc> record_witness (2)                ! Edge 2
000087cc <_zero+e8> ret                               ! Block 3
000087d0 <_zero+ec> restore
```

Figure 7: Disassembly of function `zero` in `simple.qpt`

edge markers written to the same buffer are usually only one byte long, but the SPARC architecture does not have unaligned stores. Thus, the register in question is moved to `%g7` and repeatedly byte-stored to the buffer and right shifted. `Record_witness`(n) simply writes byte n to the buffer to record a particular edge having been traversed.

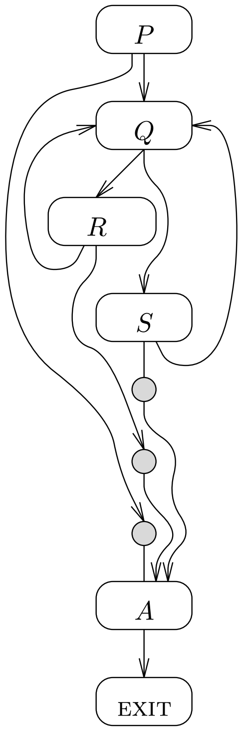
In general, instrumentation code for an edge is always written between the destination block and its predecessor. If the edge from the preceding block, called the "fall-through edge," is instrumented, then the instrumentation code for that edge is always placed first. This prevents an extra branch to instrumentation code from ever being required on fallthrough edges. All other edge instrumentation code is branched to, however, and all but the last edge must again branch to enter the destination basic block. For this reason, the last `add` instruction in a basic block is usually moved forward into the delay slot of the outgoing branch. Figure 8 illustrates the typical placement of instrumentation code for a call vertex A with three predicate predecessors.

Buffer checking will not be covered in detail, for, as we will see in the next section, it is no longer required when performing multi-tasking traces. It suffices to say that the reserved registers in the SPARC architecture are a large advantage for buffer checking, as they allow buffer checks to be placed anywhere with equal cost. On other architectures which use register stealing, one must attempt to place buffer checks only in regions where one has already "stolen" the relevant registers to record a witness or register value. However, a tremendous disadvantage of SPARC is that it is a condition code architecture. In places where condition codes are live across buffer checks, the code to check the buffer without using a comparison or conditional branch is horrendously expensive.

3 Extending *qpt* to a multi-tasking environment

3.1 Design Motivation and Overview

The system described in this section was implemented on a Sparcstation 1 with 16 MB of memory, running 4.4 BSD. The instrumented executables were generated by a modified version of *qpt* version 4.3, and made use of special kernel modifications to support tracing. The approach taken is fundamentally based on that of Borg et. al. and Chen. Every aspect of the system was developed with the goal of maximizing speed of traced execution. The performance penalties associated with gathering multi-tasking address traces can




```
if (P)
  while (Q ? R : S)
    ;
A;
```

```
or %g0, n, %g7
stb %g6, [0+%g6]
add %g6, 1, %g6
```

Figure 8: Actual placement of instrumentation code for incoming edges to call vertex A.

roughly be broken into two categories: Executing extra instructions required for tracing (and suffering the corresponding cache misses), or managing and consuming the trace data. Abstract execution, as we have seen, addresses both problems by minimizing the number of trace markers which need to be recorded. The challenge, then, was to coordinate shared use of the trace buffer and maintain or enhance the benefits of abstract execution.

Because *qpt*-instrumented executables often sequentially write the values of several registers directly to the trace buffer, any combination of bytes could potentially be written to the buffer. Using an escape sequence to flag a context switch in the trace buffer is therefore impossible without the use of run-time checks. Though witness numbers are assigned at instrumentation time, and could simply be chosen never to conflict with the chosen escape sequence, register writes would need to be checked against the escape sequence. The actual check could in theory be performed with only two instructions. However, again, as SPARC is a condition-code architecture such checks would be much more expensive with live condition codes. To avoid this problem, the tracing system developed for this work simply records context switch information in a separate buffer, and stores it in a separate file which can then be used to disambiguate the multi-tasking traces.

The full tracing procedure chosen for this work, then, is illustrated in Figure 9 on the following page. To avoid the overhead of copying trace data at each context switch, the same global buffer is mapped into all traced processes and the consumer process. These processes all share a single value of register `%g6`, and use it as a pointer into the trace buffer. `locore.s`, the interrupt and exception handler, is modified to record context switch information in a special schedule buffer. The schedule buffer is located in kernel memory above the user stack. Its first free location is pointed to by a kernel global variable `trbuf_sched_ptr`. When either the trace buffer or the schedule buffer overflows, the consumer process can store the trace data to a file, or pipe it to another process.

Currently, the consumer program is called *bdump*. *Bdump* takes two arguments: the first is the name of the file in which to store the trace data, and the second is the name of the file in which to store the schedule data. As a special case, if the first argument begins with a "`|`" symbol, it is interpreted as the name of a command through which to pipe the trace output, rather than as the name of a local file. The

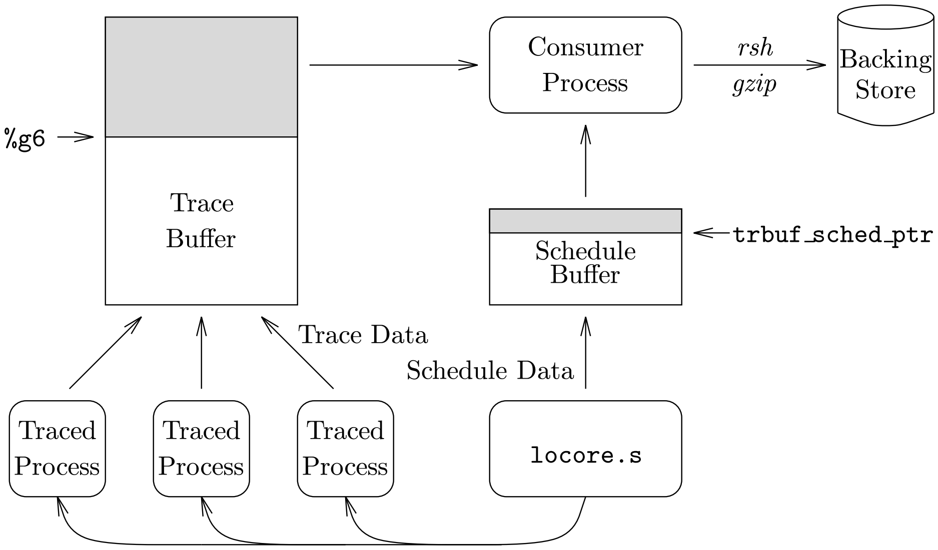


Figure 9: Overview of the approach.

best performance results were generally obtained by piping the output of *bdump* over the network through the *rsh* command, and compressing and storing the data on a DEC Alpha workstation with the GNU tool *gzip*.

3.2 Implementation Specifics

The following subsections give the implementation specifics of the system, and describe the interface to the kernel portion of the system. Section 3.2.1 explains the setup procedure required for mapping the trace buffer into a traced procedure's address space, and also the steps required for a consumer process to register itself. Section 3.2.2 discusses the special steps taken during user/kernel and kernel/user transitions which ensure that no ambiguity is introduced with context switches. Finally, Section 3.2.3 describes the method used for detecting buffer overflows and consuming the buffer.

3.2.1 Mapping the Buffers

The Sparcstation VM structure is pictured in Figure 10. The hardware requires that the most significant three bits of any valid virtual address be the same, which leaves a 3 Gigabyte gap in the middle of the address space. The lower 512 Megabytes generally contain a program's text and data, the upper 128 Megabytes are used to map the kernel into every context, and the user stack is located immediately underneath the kernel. The global trace buffer is always mapped into the address space of processes immediately below the 3 Gigabyte dead zone. No reasonable program designed to run on a Sparcstation would attempt to set its break value[11] at 512 megabytes. The only danger of a program making specific use of virtual addresses at the location of the buffer is for memory allocated through use of the `mmap` system call. However, `mmap` by default starts allocating memory around 32 megabytes, and thus could only fail if the user explicitly requested virtual addresses that conflict with the trace buffer location.[12]

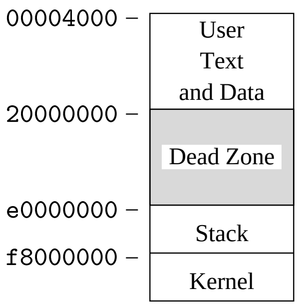


Figure 10: Sparcstation VM

The trace buffer itself is implemented as a device driver, which is called `/dev/trbuf`. When the device is first opened, it reserves a portion of non-pageable memory for the global trace buffer. The system currently

[11] i.e. the first address beyond the heap, usually only changed by `malloc` using the `brk` or `sbrk` system call.

[12] If it really becomes necessary to trace programs which insist upon using `mmap` to allocate memory before the Dead Zone, the buffer could easily be moved to another location such as the lowest possible stack address.

uses a 1 MB buffer, though this size can easily be changed at compile time through modification of a constant TRBUF_SIZE. When the device is first opened, it also initializes a kernel global variable trbuf_g6 to contain the value 0x20000000 − TRBUF_SIZE. The buffer can then be mapped into processes with one of two ioctl system calls—one for traced processes, and one for the consumer process. In order to be able to identify traced processes easily, these calls modify the value of a field called p_trace_stat which has been added to the portion of the proc structure that is zeroed upon process creation.

When instrumented executables begin execution, they execute the call

```
ioctl (fd, BIOCMAP)
```

before jumping to the first instruction of the untraced program. (fd here is the file descriptor number corresponding to the trace buffer.) This maps the trace buffer into the process's address space. In addition, the p_trace_stat field is logically OR-ed with the value PRTRC_TRC, to indicate to various parts of the kernel that the process should receive special treatment such as sharing the value of %g6 with all other traced processes. More detail will be provided about the use of this flag in the following two sections. When a consumer process begins execution, it must map the buffer with the call

```
ioctl (fd, BIOCSMAP, &bufinfo).
```

(bufinfo is simply a structure which gets the start values of the schedule and trace buffers.) This call also maps the trace buffer into the calling process's virtual address space, but sets the PRTRC_ANA flag in its p_trace_stat field. This flag also indicates that %g6 should be shared with this process, but does not trigger any of the other special behavior required for a traced process.

The Sparcstation MMU performs hardware table walking on TLB misses. However, it only fills the TLB with Page Table Entries from 128 Page Map Entry Groups (or PMEGs) of 64 PTEs each. Since the schedule buffer is written by the kernel in locore.s with traps disabled, a PMEG miss would put the processor into error mode, and abort execution of the kernel. The schedule buffer is therefore allocated in kernel memory above the stack, where the VM code locks PMEGs and guarantees their continued existence. Though protection on the schedule buffer's pages could have been changed to allow user-level access, doing so would have required short-circuiting the kernel virtual memory code, and given how small schedule data is in comparison with trace data, this was not worth the effort. Instead, schedule data is simply read from /dev/trbuf using ordinary file reads.

3.2.2 User/Kernel Transitions

Whenever the kernel is entered from user mode and either PRTRC_TRC or PRTRC_ANA is set in the current process's p_trace_stat field, locore.s has been modified to store the value of %g6 in the kernel global variable trbuf_g6. Before returning to user mode, the return from trap code again checks p_trace_stat, and if the process requires the shared value of %g6, then that register is restored from trbuf_g6 rather than from the value in the trapframe. Additionally, before returning to user mode, if the schedule buffer is nearly full then the trap code is reentered with a new, fake, trap value called T_SBOF. This causes traced processes to block as if the trace buffer had overflowed. The mechanism employed is identical to the fake T_AST trap which is used to force a reschedule when exiting the kernel. However T_SBOF takes precedence over T_AST since it will likely affect the next process to be scheduled.

The first process to open /dev/trbuf and perform a BIOCMAP ioctl, then, returns from the system call to find the address of the trace buffer in register %g6. Any subsequent traced process will find that %g6 points to the first free byte of the buffer. Regardless of what other processes have been scheduled since a traced process last ran, that process will always have a pointer to the first free byte of the trace buffer in %g6. When the trace data is stored by the consumer process, the consumer process can simply point %g6 back to the beginning of the buffer. The problem of disambiguating the concatenated traces is solved by keeping the schedule buffer.

The schedule buffer is actually written at the end of cpu_swtch, the routine that finds the next process to run and actually performs the context switch. A global word trbuf_pid is used to cache the value of the last traced process scheduled. If the new process is traced and has a different process id from the previously scheduled traced process, or if a buffer overflow has occurred since the last traced process was scheduled, then a three word marker is written to the schedule buffer, containing the new process id, and the values of

%g6 and the PC in the new process's trapframe. As we will soon see, instrumentation code which causes the buffer to overflow is rolled back, and consequently %g6 does not always hold the value 0x20000000 on buffer overflow. Whenever it rewinds %g6, then, the user-level consumer process is expected to add to its output stream a fake schedule marker with the actual value of %g6 at overflow.

Unfortunately, the situation is complicated by the possibility of exceptions or interrupts occurring during the execution of instrumentation code. If a value has been written to the buffer, and a trap or interrupt occurs before the increment of %g6, then information will be lost if a different process is scheduled upon return to user mode. When an interrupt or exception occurs, therefore, the text of the current program must be examined to determine if instrumentation code that makes use of register %g6 has just been suspended. If so, since %g6 is always incremented last in instrumentation code, the PC is simply rewound to the first instruction of the instrumentation code.

The situation is, of course, complicated still further by the fact that the instructions one wishes to examine are not necessarily in resident pages, or do not necessarily have valid PMEGs currently allocated for them. Since the modified trap handler code must obviously run with traps disabled, one must be very careful in examining the user text segment. In particular, the reason for entering the kernel could actually be a text fault, which may or may not have been caused by instrumentation code spanning two text pages. However, two facts allow us always to recognize instrumentation code at trap time: First, all instrumentation code is straight line code. Witness code does contain a branch, but the branch is delayed, and its target instruction is only executed after the increment of %g6. Second, since we always roll back instrumentation code on a user/kernel transition, the first instruction of an instrumentation code sequence will always be resident if any other instruction in that code sequence is interrupted or causes a trap.

The method employed, then, is to look not at the trap PC, but at the previous instruction. If that instruction is not currently accessible, then the trap instruction is the target of a branch and we can safely assume that the buffer and %g6 are in a consistent state. If, on the other hand, the previous instruction is resident and belongs to the middle of an instrumentation code sequence (which we can easily recognize by the fact that it makes use of reserved register %g7 as a temporary register),[13] then we continue scanning backwards, looking for the the first instruction of the sequence, aborting the process at any time if we cross a page boundary and find the page invalid.

3.2.3 Buffer Overflows

Qpt, when tracing single processes, is constantly forced to check for space left in the buffer, and branch off to write the buffer to disk in *flush_buffer* whenever there is not enough space left. This can be expensive, particularly when the checks must occur in places with live condition codes, where the buffer check code cannot make use of conditional branches. In multi-tasking mode, however, traced executables are not responsible for storing their own traces. There is no reason, therefore, to make them responsible for checking free space in the buffer.

Whenever the trace buffer overflows, it causes a memory data fault to occur at address 0x20000000. This is therefore used to detect buffer overflows. Whenever such a fault occurs, and the process causing the fault has PRTRC_TRC set, the memory fault code invokes function trbuf_block with a non-zero argument to block all traced processes, including the currently running one, until the buffer has been processed and reset by the consumer process. When trbuf_block is called, a global kernel variable trbuf_isblocked is set to non-zero, and all runnable traced processes are pulled off the run queue and placed onto a fake sleep queue which is inaccessible to the normal wakeup function. At this point, the process that caused the trap also sleeps, and when it is woken up will return from the memory fault as if a simple PMEG miss had occurred. If, while processes are blocked, any traced processes which were sleeping at trbuf_block time are woken up, the wakeup is recorded by logically OR-ing PRTRC_WAK with the p_trace_stat field of the process's proc structure, but the process is not placed back on the run queue until after the consumer process is finished.

[13] In reality, the previous instruction may be a branch, preceding the increment of %g6 in a *record_witness* sequence. We therefore look back one instruction further (checking first that this will not cause a fault), whenever we find a branch.

3.3 Comparison

One of the significant disadvantages of Chen's system over its predecessor was the need to copy trace data multiple times before consuming it. Borg et. al., on the other hand, were able to map the same trace buffer into all processes, which they did by sharing the buffer pointer in a register between all processes. Fortunately, the SPARC ABI by reserving `%g5...%g7`, allowed the system developed for this study to map one buffer globally. It should be noted, however, that though no compiler generated code makes use of these registers, a few of the hand-written assembly-language functions in the BSD *libc* library actually did, and needed to be recoded slightly.

A large disadvantage of both Borg's and Chen's systems is the amount of time spent consuming the trace. Consuming trace data in a cache simulation is at best 10 times slower than producing the data. However because of the sheer size of the data in the two previous systems, storing it to disk or tape was even slower. An uncompressed standard address trace requires 5 bytes per trace item. 4 bytes are required to record the address of the memory reference, and one byte is required to record the type (Load, Store, or Basic Block Entry) as well as the length of the reference. In the Titan system, the 32 Megabyte buffer filled up in less than one second of untraced execution. By comparison, abstract execution produces on the order 1–5 megabytes of uncompressed trace data per second of untraced execution. In addition, *Qpt*'s trace data is quite compressible, and can typically be reduced by another order of magnitude by the GNU *gzip* compression utility. Thus, in this system we can execute instrumented programs and store the traces with only slightly higher overhead than the previous systems spend simply executing the instrumented executables.

4 Results

The first round of experiments show that user-level execution time of traced executables, as estimated by the Unix *time* command, is roughly three to four times greater than that of untraced executables. The global trace buffer fills up at a rate of approximately two-thirds of a byte per memory reference, and can be forwarded across the network, compressed, and stored at a rate of 350 Kbytes per second. As a result, CPU-intensive programs perform the worst under this system, experiencing a slowdown of just under 18 times. This compares very favorably with Borg's and Chen's systems, in which the reported slowdown is slightly less, but does not include the time to store or consume the trace—typically an order of magnitude greater than traced execution time in their systems. I/O-bound and interactive programs, on the other hand, did not experience any significant slowdown during execution, and storing the traces only slowed them down by a factor of 3.

In order to characterize performance of the system, two sorts of programs were traced: interactive and non-interactive programs. Performance of interactive programs is difficult to characterize quantitatively, for running time depends on user input, and latency is as important a factor as throughput. Several simple X-windows sample programs were instrumented and traced simultaneously, and their performance was found to be comparable to untraced versions, save for occasional interruptions during buffer processing. This paragraph, for instance, is being typed using an instrumented copy of the simple X-windows text editor *xedit*, while an instrumented version of the follow-the-cursor demo program *xeyes* is also running. During traced execution, the program seems just as fast as normal. For every 40 or so characters that are entered into the buffer, there is an annoying pause for a second or two as the buffer is sent over the network and stored. However, this is not too much worse than auto-saves by the *emacs* text editor, which may also take a second or two for large buffers saved onto NFS mounted partitions. In short, it is quite reasonable to expect to trace interactive X applications without the user loosing patience and no longer providing a realistic input stream.

The trace described in the preceding paragraph ran for 18 minutes as the author tested the scrolling in *xedit*, played with the *xeyes* demo, and typed in and edited the paragraph. The resulting *gzip*ped trace file is approximately 5 megabytes, and the schedule file is 32 K. These numbers reflect the fact that both programs spend most of their time sleeping, waiting for input. Non-interactive programs clearly presented a much more stressful test of the system. Figure 11 summarizes the results for three programs, sorted in order of decreasing CPU-utilization. The *Li* benchmark is based on the *xlisp* lisp interpreter included with the SPECINT92 benchmark suite, and measures the time required to solve the 7-queens problem. The *Gcc*

Program	Untraced Time	Dummy Time	Traced Time	Memory References	Trace Size
li	0:36	1:51	10:30	275 M	28 M
gcc	0:13	0:35	3:50	144 M	13 M
egrep	0:12	0:16	0:41	9.7 M	1.4 M

Figure 11: Traced and Untraced Execution Times

benchmark listed is another subset of the SPECINT92 benchmark suite, and measures the time necessary for *gcc* version 1.35 to turn one of its own pre-processed source files, `1stmt.i`, into optimized 68000 assembly code. Finally, *egrep* is simply the *egrep* utility which came with 4.4 BSD, searching a 5 megabyte input file for a pattern which the file does not contain.

There are three times reported for each program. Untraced time is the execution time of the uninstrumented program on the chosen input file. Dummy time is the running time of the instrumented executable without the time required to store the trace. It was measured by tracing the program as usual, but saving the trace and schedule buffers to `/dev/null` rather than to an actual file. Finally, traced time is the time required to run the instrumented executable with a 1 megabyte trace buffer, and pipe the trace information through *rsh* to a DEC Alpha workstation, where it is compressed with the command `gzip --fast` and stored to a local disk. Each test was run at least three times, and the numbers reported for all three cases are the most often encountered execution times, which were always also the best times. Occasionally much higher execution times were encountered. For example, the first execution of *gcc* took 45 seconds. Most of the extra execution time appeared to take place before the introductory version message was printed out. Since the executables were all on NFS-mounted partitions, this and other delays could be explained by the fact that throughout the experiments a networking project was often flooding the network with packets. For this reason, it seems reasonable to discard high execution times in determining performance for both traced and untraced execution.

As expected, tracing overhead is proportional to the amount of time spent in user-level execution. Both *li* and *gcc* spend over 90% of their execution time in user level, as estimated by the *time* command. *Li* has negligible input and output files, and performs the worst. `Gcc` has a small amount of I/O, namely a 110 K input file and a 90 K output file. *Egrep*, on the other hand, spends 90% of the time executing in the kernel or waiting for I/O and performs considerably better. Measuring the performance of CPU-bound benchmarks in the system basically gives us an upper bound on tracing overhead. Programs with lower amounts of user-level CPU utilization execute faster to some extent because by Amdahl's law a smaller percentage of execution time is affected by the tracing, but another significant part of the improvement is that ratio of the rate of trace data creation to consumption is lower. Fortunately, however, CPU-bound programs are also those whose analysis is least affected by time dilation. Because of their infrequent context switches and ability to scale well with raw integer performance of processors, they are also those least interesting to trace in a multi-tasking context.

During traced execution of a single CPU-intensive process, the traced process typically consumed only 10–20% of the CPU time measured by the *ps* command. *Bdump* typically consumed the same or slightly less than the traced program, while *rsh* always consumed between 30 and 40% of the CPU. In order to try to obtain higher CPU utilization, a second consumer process, *bdump2*, was run in parallel with the traced processes. *Bdump2* checks the status of the buffer every second or whenever the buffer fills up, whichever occurs sooner. When *bdump2* finds more that 16 Kbytes of unsaved data in the buffer, it immediately writes the data to its output. The X-windows applications benefited greatly from *bdump2*, as they blocked for much shorter periods of time. The three non-interactive benchmarks, as well as an extended *egrep* benchmark, improved by less than 5%, however. This suggests that the performance limiting factor is really network throughput.

5 Future Work

5.1 Speed Optimizations

The limiting factor in performance of traced executables is clearly the time required to store the trace data when the buffer has overflowed. There are two approaches to improving this time: Reducing the size of the trace data, and optimizing the processing of that data. The first implementation of this tracing system leaves open possibilities for improving both of these factors. Let us first consider methods for reducing trace size. The basic premise of Section 2.2 was that information about a program's execution path could only be recorded through use of unique markers, emitted whenever specific regions of code were executed. However, since trace regeneration code is specific to each function, abstract execution can easily allow certain functions to be hand instrumented, or instrumented using a different algorithm. We have already seen that functions such as `setjmp` and `longjmp` can receive special treatment. As long as the regeneration code for a particular traced function guarantees the input stream to be at the correct position before procedure calls and returns, any method can be used for tracing that procedure.

Consider, for instance, the function `bcopy`, which copies a region of memory to a new location. The code for `bcopy` contains quite a few conditionals in order to handle alignment and overlap correctly, and a main copy loop to move data in word-long quantities. A simple, word-aligned, non-overlapping `bcopy` of 1 Kbyte generates 268 bytes of trace data. Since the path of execution and the addresses of all loads and stores are completely determined by the three input arguments, however, the optimal number of trace bytes required for `bcopy` is only 12—more than 20 times fewer. Often used and highly deterministic functions such as `bcopy` could be recognized by name and specially instrumented so as to minimize the amount of data written into the trace buffer.

Even without hand optimizing the instrumentation of a procedure, there are improvements that can be made over simple edge and block profiling. Goldberg[Gol91] has shown that when instrumenting code for counter-based profiling, loops can often be recognized automatically, and the induction variable used to count iterations of the loop. A similar optimization can be used for tracing loops with no impossible instructions. By either using the induction variable of the loop or adding in a counter, instrumented code could record only the number of iterations of an inner loop, rather than a marker for each iteration.

Of course, when tracing executables that have been instrumented by hand or with the loop optimization, simply storing the PC at context switch time will no longer be enough information to determine the exact location of the context switch. However, there is still one register reserved by the SPARC ABI which *qpt* does not make use of, namely `%g5`. If `%g5` is always used as the loop counting variable, and its value is recorded in the schedule buffer at context switch time along with `%g6` and the trap PC, then regeneration code can determine in exactly which iteration of the loop a context switch occurred.

As for optimizing buffer processing, piping the trace data through *rsh* results in an unnecessary copy of the buffer simply for the pipe. The networking code should undoubtedly be integrated into the consumer process, so as to manipulate the data from the buffer directly. The most important factor, however, is clearly network throughput. Fortunately, however, there is much room for improvement. As a test, the Sparcstation upon which most of this work was done was booted with an optimized, non-tracing kernel and connected to an isolated 10BaseT concentrator with only one other machine, a PC, on the network. Tests using *ttcp* revealed throughputs of only 400 Kbytes per second. For comparison, the PC used in the test was able to achieve throughputs of over 900 Kbytes over a shared network to a Sparcstation ELC running SunOS 4.1.3. The network problems may be hardware related, or could be a result of poor device specific code in the BSD kernel, which was only recently ported to the SPARC architecture. In either case, however, improving network performance would also improve tracing performance.

5.2 Tracing the Kernel

Clearly the tracing system developed for this work will be most useful when it is extended to trace the kernel. There are several complications involved with tracing kernel activity, however. First, the kernel must be able to run in both traced and untraced modes. When booting and processing buffer overflows, for instance, one cannot simply suspend execution of the kernel as one does for user processes. Chen turns kernel tracing on and off by checking a flag in each instrumentation code sequence. Such a scheme, however, is too expensive for the performance goals of abstract execution. Recording most witnesses takes only three instructions in *qpt*. Checking a flag without disturbing condition codes would consequently add significant overhead. Fortunately, the Sparcstation MMU will allow the same physical page to be mapped into an unlimited range of virtual pages with only one wasted PMEG. Tracing could therefore be redirected to a throw-away buffer of a single physical page, and the memory data fault handler modified to wrap the buffer pointer on overflows.

The kernel also contains a great deal of code that could more safely be instrumented with relocation information. Thus to minimize the number of procedures that need to be hand-instrumented, it would be best to use link-time instrumentation rather than direct instrumentation of `a.out` executables as *qpt* does. Modifying *qpt* to support link-time instrumentation is not an unreasonable task, and would also benefit text segment size in instrumented user-level executables.

As in Chen's system, much of the code in `locore.s` that runs with traps disabled will need to be instrumented by hand. Such code must also perform buffer checks, as a data fault on an overflow would put the processor in error mode. Furthermore, such code is often concerned with register windows, either because a trap has occurred and all the register windows could potentially be in use, or because various stacks are being manipulated and switched between, or because the kernel is preparing to return from a trap. In these cases, heavy use is made of the general-purpose `%g` registers. `%g5...%g7` are, in fact, actually used more often than the lower `%g` registers. Thus such code would probably best be instrumented minimally by hand, using whatever registers are available, with a corresponding C function written to decode the trace. Before traps are enabled, however, `%g6` and the trace buffer must always be in a consistent state.

A third potential problem with tracing the kernel will be the performance impact on programs which spend a high proportion of time in the kernel, as these were the programs with the best tracing performance, and the ones most interesting to study. However, the Monster[NUM92] study described earlier found that more than half the time spent in the kernel during execution of *ld* was spent in the idle loop waiting for I/O to complete. Of the remaining time, a significant percentage was spent in the `bcopy` and `bzero` routines. However, these can be hand instrumented as described in the previous section so as to minimize the size of trace data produced and the overhead of instrumentation.

Another challenging aspect of tracing the kernel will be avoiding gaps in the trace data when the buffer overflows. Abstract execution generates exact traces from a very specifically encoded trace file. A single corrupt byte will invalidate the remainder of a witness file. When the trace buffer overflows in the kernel, though, one cannot simply preempt the current kernel thread to schedule a user-level consumer process. With enough RAM in the machine, however, it may be possible to guarantee no kernel overflows before waking up a thread or invoking the C trap handler function after a trap. Alternatively, the data might be stored directly by the kernel, though such a scheme would undoubtedly have other undesirable consequences.

6 Conclusion

Previous work has shown the importance of address traces to analyzing system performance. As raw CPU speed continues to increase, not all applications are benefiting linearly from the improvement. In particular, memory- and I/O-intensive applications, client/server applications, and operating system kernels risk seriously lagging behind the performance of applications such as those in the high-profile SPEC benchmark suite. Unfortunately, the applications whose performance most seriously needs analysis are also those most difficult to trace; since their behavior depends upon the completion of asynchronous events, the overhead of tracing will make other events appear to occur more rapidly.

Other systems for multi-tasking tracing have slowed execution down by a factor of 10 or 15, and required 100 times the untraced execution time to process the trace whenever the buffer overflowed. By using the technique of abstract execution, multi-tasking traces have been gathered in this study with a worst case execution slowdown of only 3–4 times, and combined execution and trace storage has been performed in at most 18 times the original execution time of CPU-intensive applications, and in under 5 times the original execution time of I/O-limited programs. While other systems have required extra RAM to use trace buffers of 32 megabytes or more, all results were obtained on a 16 megabyte workstation with only 1 megabyte reserved for the trace buffer.

Many optimizations can still be made to improve performance even further. To begin with, the BSD operating system on the Sparcstation used for the experiments had unexplainably low TCP performance, which seriously limited the rate at which trace information could be stored across the network. If TCP performance were doubled, one could expect the traced performance of the slowest CPU-intensive benchmarks nearly to double. We have also seen that the technique of abstract execution can be enhanced by generating a constant amount of trace information for certain program loops, rather than a byte for each iteration. Given the large performance gain of the current system over any previous ones, and the possibilities for further improvement, realistic software tracing of I/O-bound programs no longer seems an impossibility.

Acknowledgments

Most of this work was performed as a senior for the 1993–94 academic year. We would like to thank Margo Seltzer for giving us use of a machine on which to run *kgdb*, and often answering BSD questions. This study made use of the QP/QPT Performance Tools developed by James R. Larus at the University of Wisconsin-Madison.

References

[Aga89] Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Kluwer Academic Publishers, 1989.

[BKW90] Anita Borg, R. R. Kessler, and David W. Wall. Generation and Analysis of Very Long Address Traces. In *Proceedings The 17th Annual International Symposium on Computer Architecture*, 1990.

[BL91] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. Technical Report 1031, University of Wisconsin, September 1991.

[Che93] Bradley Chen. Software Methods for System Address Tracing. In *The Fourth Workshop on Workstation Operating Systems*, 1993.

[Che94] Bradley Chen. Memory Behavior of an X11 Window System. In *The Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[CK] Robert F. Cmelik and David Keppel. *Shade: A Fast Instruction-Set Simulator for Execution Profiling.* Sun Microsystems Laboratories, Inc.

[Gol91] Aaron J. Goldberg. Reducing Overhead in Counter-Based Execution Profiling. Technical Report CSL-TR-91-495, Stanford University, October 1991.

[Lar90] James R. Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. Technical Report 912, University of Wisconsin, February 1990.

[LD91] Harry R. Lewis and Larry Denenberg. *Data Structures & Their Algorithms.* Harper Collins, 1991.

[NUM92] David Nagle, Richard Uhlig, and Trevor Mudge. Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures. Technical report, University of Michigan, 1992.

[Ous90] John K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *The Proceedings of the USENIX Summer 1990 Technical Conference*, 1990.

[Ram] Rambus, Inc. *Rambus Technology Guide.*

[Smi91] Michael D. Smith. *Tracing with* pixie. Stanford University, 1991.

[SPA] *The SPARC Architecture Manual, Version 8.*

[Wal92] David W. Wall. Systems for Late Code Modification. In *Code Generation—Concepts, Tools, Techniques.* Springer-Verlag, 1992.