Low Perturbation Address Trace Collection with Simple Hardware Performance Monitors[1]

Russell Daigle[2], **Chun Xia**[3], and **Josep Torrellas**

Center for Supercomputing Research and Development
Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801

Abstract

While address trace analysis is a popular method to evaluate the memory system of computers, getting accurate traces of operating system, multiprogrammed, and parallel workloads in multiprocessors is hard. This is because the true behavior of these real-time loads can be easily altered by the trace collecting activity. To minimize perturbation, it is usually necessary to use trace-gathering hardware performance monitors. Unfortunately, these devices often have limitations. For example, they often only capture physical addresses, can only collect references that miss in on-chip caches, and can only monitor a very small time window.

In this paper, we show how to improve the capability of such devices. We present a simple methodology to instrument operating system and applications to transfer a wide variety of information to the trace-gathering hardware with little perturbation. For instance, the information transferred includes the virtual to physical address mapping or the sequence of basic blocks executed. Each piece of information is transferred as cheaply and non-intrusively as one or a few machine instructions with one or a few cache misses. With this approach, instrumenting every basic block in the operating system and application to reconstruct the sequence of instructions executed causes only a 1.6-2.8 execution slowdown. This is in contrast to a 10-times slowdown for fully software-based instrumentation. Furthermore, the behavior of the operating system is largely unaffected.

1 Introduction

Address trace analysis is widely used for performance evaluation of computer memory systems. By analyzing the references issued by the processor, trace-based evaluations can give accurate insights into cache performance, synchronization activity, and memory system contention. However, the gathering of the traces must not significantly perturb the system being analyzed. Otherwise, the evaluation would be inaccurate. Such perturbation is usually not an issue when gathering traces of single applications in uniprocessors because it is usually unable to change the references and their order much. However, non-intrusively obtaining traces of operating system, multiprogrammed, and parallel workloads in multiprocessors is particularly hard due to the real-time nature of these workloads. In these cases, any interference due to the gathering of address traces can severely alter the references and their order. For example, if tracing significantly slows down the operating system, the relative timing of asynchronous events such as timer interrupts will change and the traces will be inaccurate.

In these workloads, the approach to trace collection that involves the least perturbation entails the use of trace-gathering hardware devices like data acquisition systems (DAS) or hardware performance monitors. Unfortunately, while we would like such devices to gather different types of information, they are very limited. Indeed, they usually capture physical addresses only, which makes the data hard to analyze. Furthermore, they can only capture the references that miss in on-chip caches or, if the devices are attached to the bus, only the references that access main memory. Finally, these systems have a limited trace storage capacity, which allows them to monitor

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436, RIA MIP 93-08098, MIP 93-07910, and MIP 89-20891; NASA Contract No. NAG-1-613; and a grant from the University of Illinois Research Board.

[2]Currently with Tandem Computers Inc., Cupertino, California.

[3]Currently with Sun Microsystems, Mountain View, California.

only very small time windows. As a result of all this, only limited performance evaluation studies can be performed.

For such common performance monitoring systems, this paper presents a general method to gather traces of operating system, parallel, and other real-time workloads in a manner that is simple and little intrusive. The method consists of augmenting the code with dummy load or store machine instructions. These instructions issue references that bypass the caches and are either TLB-unmapped or have a virtual to physical address translation that we can easily find out. The addresses accessed by these instructions are captured by the performance monitor and interpreted by the trace postprocessing program. Using this approach, the processor can pass information to the trace postprocessing program with low perturbation. In this paper, we also describe two prototype multiprocessor implementations of this approach. Finally, we measure the perturbation caused by the instrumentation and show that, even with much instrumentation, the behavior of the operating system is largely unaffected.

The remainder of the paper is organized into four main sections: Section 2 discusses previous work, Section 3 describes the types of systems considered, Section 4 presents the methodology of our approach, and Section 5 evaluates the accuracy of our approach.

2 Previous Work

Address trace collection systems can be classified into hardware-based and software-based. Hardware-based systems rely on trace-gathering hardware devices, while software-based systems are based on software instrumentation of the code to be traced.

The two main advantages of hardware-based systems is that they collect complete traces of the workloads and that they do so with little or negligible perturbation. By complete, we mean that they capture the activity of the applications, operating system and background daemons. However, they can be difficult to use. Indeed, often, they only collect physical addresses, record only the accesses that miss in the caches of the machine, and have very limited trace storage capabilities. In addition, they tend to be expensive and non-portable. Several trace-collecting hardware devices have been used, especially in uniprocessors. For example, Alexander *et al* [1] used a logic analyzer to gather address traces from a NS32016 microprocessor. This system can observe all the accesses issued by the processor because the processor monitored does not have on-chip caches. In general, systems based on performance monitors tend to work with samples because they have a very limited trace storage capacity. The two systems described in this paper [18, 19], however, can generate continuous and complete traces. Finally, there are hardware monitors that simply count events. One example is Monster [14], a logic analyzer carefully programmed to count cache misses, TLB misses, write buffer stalls, and other events. Counters, however, only allow the measurement of a very limited number of event types.

Software-based systems, on the other hand, are attractive because they are cheaper than hardware-based systems and are more portable. However, the disadvantages are that it is very hard for them to capture a complete trace of the workload as defined above and that they perturb the system that they are measuring to some degree. While the perturbation may be acceptable for single applications in uniprocessors, it may not be acceptable for real-time workloads like the operating system, multiprogrammed, and parallel workloads in multiprocessors.

There are different types of software-based systems. One type are those that instrument the application such that, at certain points in the application, a monitoring routine is called. These points may be, for example, basic block entry points or memory access instructions. The monitoring routine saves a trace entry or increments a counter. The instrumentation is usually done on the executable file, although it can also be done on the assembly file. While there are some differences among the different tools, this broad category includes, for example, ATOM [16] and EEL-QPT [12] as well as many other older systems [7, 9, 10, 13]. The new tools have been highly tuned and run very efficiently. However, they are designed to instrument a single application. Some of them, like

ATOM, have been applied to a uniprocessor operating system. However, these are hardly the right tools to capture complete traces of a multiprogrammed parallel workload that includes applications, operating system and several daemons.

A second type of software-based systems are trap-driven simulation systems [20]. In this approach, simulations are driven by kernel traps. These traps allow the simulation of a cache as the kernel executes. Indeed, memory traps are set on addresses that are currently not in the simulated cache. When that address is accessed, the kernel traps. After trapping, the kernel increments the count of number of simulated misses, clears the trap since the address is now considered to be in the simulated cache, and sets a trap for the address displaced from the simulated cache. TLBs can be simulated in a similar way. With this approach, both operating system and application effects are considered. However, a major disadvantage of this scheme is that it cannot generate as much information as systems based on traces. Examples of this approach are Uhlig *et al*'s Tapeworm II [20] and Talluri's TLB simulator [17].

Finally, the last type of software-based system is exemplified by SimOS [15]. In SimOS, the hardware of a machine is simulated with enough detail to run an entire operating system. On top of this operating system, we can run applications. With this system, we can collect complete traces as defined above. The advantage of SimOS over the hardware performance monitors is that SimOS is much cheaper. Two problems of SimOS are that it is a very complicated software system and that it is hard to port to different platforms. In addition, the accuracy of SimOS simulations compared to simulations based on hardware performance monitors is unclear at this point. The workloads will run significantly slower on SimOS than on real hardware with a performance monitor. This is specially true if we simulate a multiprocessor system. For instance, we will see in this paper that we can collect full instruction and data traces with a hardware performance monitor with a slowdown factor of 2. It is unclear how slow the same workloads would run on SimOS. Based on [15], they are likely to have a slowdown factor that is at least one order of magnitude higher than that, and more likely even higher.

3 Systems Considered: Characteristics and Requirements

The previous section has shown that there are many types of trace collection systems. In this paper, we focus on those systems that collect complete multiprocessor traces using simple hardware performance monitors that do not perturb the system. The performance monitors considered have a trace buffer that collects addresses referenced by the processors. The address collection proceeds quickly, without slowing down the workloads at all. In this section, we first give as an example two such systems that we have built in the very recent past. Then, we list the advantages and disadvantages of this approach to collecting traces and describe the hardware and software requirements for these systems to be useful.

3.1 Examples of Systems

In the course of the past few years, we have developed two trace generation systems that have these characteristics and are connected to two bus-based shared-memory multiprocessors. The first one is connected to a Silicon Graphics multiprocessor [18], while the second one is connected to an Alliant multiprocessor [19]. In the remainder of this section, we describe these two systems.

Example 1: Silicon Graphics

This trace generation system [18] is connected to a Silicon Graphics POWER Station 4D/340 [5]. The machine is a bus-based cache-coherent multiprocessor with four 33-MHz MIPS R3000 processors. Each processor has three physically-addressed, direct-mapped caches: a 64-Kbyte instruction cache, a 64-Kbyte first-level data cache, and a 256-Kbyte second-level data cache. All caches have 16 byte blocks and are kept coherent with the Illinois cache coherence protocol. The operating system running on the machine is IRIX 3.2, which is based on UNIX System V. Each processor runs one thread of IRIX. All threads share all the operating system data and can execute most of the operating system functions.

A hardware monitor connected to the bus of the machine records all bus activity without affecting the system. Since the monitor is connected to the bus, it can only record accesses that miss in all caches. For each bus transaction, the monitor captures the physical address, whether the transaction is a read or a write, the originator of the transaction, and the number of cycles elapsed since the last bus transaction. All this information is stored in a buffer with 2 million entries. Depending on the frequency of cache misses in the workload that is running, bus transactions fill up the trace buffer in 0.5 to 4 seconds.

Example 2: Alliant

This trace generation system [19] uses a 4-processor Alliant FX/8 bus-based multiprocessor [2]. In this machine, each processor has a 16-Kbyte direct-mapped primary instruction cache. In addition, all processors share one 512-Kbyte 4-way interleaved cache with 32-byte lines. This cache receives all data accesses plus the instruction accesses that miss in the primary caches. The multiprocessor operating system running on this system is Xylem [11], a derivative of Concentrix 3.0. Concentrix is Alliant's commercial operating system and is based on 4.2 BSD UNIX. Xylem is symmetric and all data is shared by all processors.

Between the primary caches and the shared cache, we connect a hardware performance monitor [3] that can gather address traces. The monitor has 4 trace buffers connected to the four processors. Each trace buffer gathers the physical addresses of the memory locations referenced by the loads and stores from one processor. Each buffer stores up to one million events. The information stored per reference includes the physical address, a 20-bit time-stamp, a read/write bit, and other miscellaneous bits. The trace buffers typically fill in around 3/4 of a second. The traces from the four buffers are later merged. We know the order of the traces thanks to the time stamps in the traces.

3.2 Advantages, Disadvantages and Requirements

The major advantage of these systems is that trace collection proceeds without affecting the workloads that are running on the multiprocessor. The workloads run at the same speed as they would if they were not being monitored. Therefore, we can gather very accurate traces.

These systems, of course, have the shortcomings of being potentially expensive and hard to port across platforms. In addition, without any extra support, they have the following disadvantages:

- Trace collection is skipped for certain time periods. This occurs because we want to examine long traces even though the trace buffer fills up quickly. As a result, we need to repeatedly read out the contents of the buffer, save or process the traces, and then refill the buffer. Consequently, we are skipping trace collection while the buffer contents are being saved.
- It is hard to interpret the trace. Ideally, we want to use the trace to understand what parts of the code and what data structures cause the misses in the memory hierarchy of the machine or in a simulated one. This is not made easy by the fact that the trace buffer only collects addresses. In addition, it is made quite hard by two other effects:
  - The trace buffer is often unable to observe all the references issued by the processor. This may be due to different reasons. In processors with on-chip caches, it is impossible to connect the trace buffer between the processor and the primary cache. Therefore, we can only gather references that miss in the primary caches. In other systems, the performance monitor is connected to the memory bus and, therefore, can only observe accesses that miss in primary and secondary caches. For example, in the Silicon Graphics system,

accesses that hit in any of the three per-processor caches are invisible to the monitor. In the Alliant system, instruction accesses that hit in the primary cache are invisible. All this incompleteness makes it harder to extract useful information from the trace.

  – In addition, these systems gather physical, not virtual addresses. This is because, usually, the virtual to physical address translation takes place on chip. Without the virtual addresses, it is often very hard to determine what part of the code or what data structure generated a given reference or cache miss pattern.

One of the purposes of this paper is to show how to eliminate all these shortcomings cost-effectively. Our approach is as follows:

- To avoid missing workload traces while the buffer contents are being saved, we stop the workload before the trace buffer gets full. After the buffer is emptied and ready to accept new traces, we restart the workload.
- To create traces that are more readily analyzable, we annotate the trace with some dummy trace entries. These trace entries are generated by dummy accesses that bypass the caches. These accesses transfer critical information from the processor to the trace buffer to enable the partial reconstruction of the trace later on.
- Finally, to help interpret the trace, we dump the information in the TLBs or page tables into the trace. With this information and the physical addresses, we are able to regenerate the virtual addresses referenced by the processor.

4 Methodology

In this section we describe in detail the method that we use to fulfill our three requirements for a useful tracing system as described above: stopping and restarting the workload, interpreting the trace, and regenerating virtual addresses.

4.1 Stopping and Restarting the Workload

This operation can be supported with different degrees of software and hardware. In the two systems that we built, we use a different tradeoff. In the Silicon Graphics system, we use a software-only approach while in the Alliant system we rely on interrupt hardware support. In the Silicon Graphics system, at the beginning, a master process forks off the workload programs, sets up an alarm so that it can wake up at regular intervals, and goes to sleep. At regular intervals, the alarm wakes up the master process. Right after waking up, the master process checks the trace buffer. If the fraction of the trace buffer that remains empty is more than a threshold value, then the master goes back to sleep. Otherwise, the master suspends all processes, dumps the traces in the buffer to disk, and restarts the processes that were being traced. The threshold that determines whether or not the master goes back to sleep is chosen in conjunction with the alarm period so that there is never any risk of buffer overflow.

Clearly, this setup allows for the gathering of uninterrupted address traces. However, it requires some software support. First, the master process is given real time priority to ensure that, when it wants to run, it is never held back by any other process. This guarantees that the state of the buffer is checked at the correct times, preventing any buffer overflow. Second, we use a modified suspend signal that is processed immediately by all processes that receive it. Finally, instead of dumping the trace on a disk locally, the master process sends the trace to a remotely-mounted disk. In the remote machine, another program postprocesses the trace in parallel while the next segment of the trace is being generated and transferred. This setup exploits parallelism and, in addition, the activity of the postprocessing program does not pollute the caches and memory of the system under measure.

In the Alliant system, instead, we use a hardware approach. We have added a connection between the four trace buffers and the four processors. This connection can activate a non-maskable interrupt (NMI) in the processors. The way it works is as follows. When one of the four buffers nears filling, it sends a NMI to the processors. Upon receiving the interrupt, processors trap into an exception handler that we wrote and halt in less than ten machine instructions. Then, a workstation connected to the performance monitor dumps the traces in the buffers to disk or processes the traces and discards them. Once the buffers have been emptied, the NMI is cleared and the processors resume. Clearly, this technique also allows for the gathering of uninterrupted address traces. Furthermore, this is done with negligible perturbation. Indeed, we coded the interrupt handler so that, upon receiving a NMI, each processor halts in a few machine instructions.

In both the Silicon Graphics and the Alliant systems, when we halt the workloads that we are tracing, we do not halt any outstanding I/O operation. We neglect to do so to simplify the experimental system. The net result is that the system may appear to have an I/O subsystem with a higher performance than it really has. We tolerate this source of inaccuracy because we are not directly interested in I/O performance.

4.2 Interpreting the Trace

As described in Section 3.2, to interpret the trace, we need to augment the data in the trace buffer with some extra information that help us identify what parts of the code and what data structures cause most of the trace activity. In addition, such extra information must also help us regenerate the virtual addresses of the traces. This information is encoded in dummy trace entries that are induced by some instrumentation code that we add to the workload being traced.

In this section, we first discuss the information that we suggest including in these dummy trace entries to be able to interpret the trace. Then, we propose a technique to generate these dummy trace entries.

4.2.1 Information to Include in the Trace

There are several pieces of information that we suggest including in the trace data in these systems. These pieces of information will help us interpret the trace later.

To understand what parts of the code generate the trace activity, we need to know, at every point in the trace, what subroutine is being executed. This involves adding instrumentation at the beginning of each subroutine and, if the program that we are tracing is the operating system itself, right after each subroutine call instruction. The latter instrumentation is needed in the operating system to determine what subroutine is executing at any given point in time. This is because subroutine calls and returns in the operating system are not fully nested. For example, given two subroutines *Caller1* and *Caller2* that have calls to *Callee*, oftentimes *Callee* is called from *Caller1* and returns to *Caller2*.

Often, we need as fine a granularity as to know what basic block is executing at every point in the trace. In that case, we need to instrument the beginning of each basic block.

It is useful to separate application from operating system references. In this case, we need to record every time that we enter and exit the operating system. This requires instrumenting the exception handlers in the operating system. While monitoring operating system entries is easy, monitoring operating system exits is difficult. The reason is that, often, the operating system exit code returns to the operating system instead of the application. In fact, the number of times that the entry code is executed and the number of times that the exit code is executed are different. Therefore, to determine if we really return to the application, we need to check the value of a status register before the return instruction.

In workloads with several processes, we may want to record the references and misses on a per-process basis. In this case, we need to know what process is running on each processor at any time. This involves instrumenting the scheduler routine in the operating system to read the PID of the running process.

Finally, to interpret a trace, it is very important to be able to regenerate the virtual addresses of the accesses. For example, to determine whether an application reference is an access to the instruction or to the data cache, we need to know its virtual address. This suffices because instructions and data are allocated in different ranges of virtual addresses. Furthermore, we often need virtual addresses to find out what instruction or data structure is responsible for a given access or miss.

As indicated before, the trace buffer stores physical addresses only. Therefore, we must save in the trace buffer the virtual to physical page translations for all running processes. There are two ways of doing this. One way is to intercept the virtual to physical page translations as they are loaded in the TLBs and save them in the buffer. The second way is to intercept the virtual to physical page assignments in hard page faults and save them in the buffer. Obviously, the first alternative involves saving more information as the program runs.

In the Silicon Graphics system, we used the first alternative because we also wanted to study the performance of the TLBs. We first wrote a system call to dump the TLBs when the workloads were initially started and every time they were restarted. Then, we instrumented the assembly routines in the operating system that modify the TLBs. This allowed us to inform the trace buffer of any change made to the TLBs. In the Alliant system, instead, we monitored hard page faults. Every time that a page fault for the process occurred, we dumped the new translation into the trace buffer. In any of the two systems, every time that we need to save an entry, we save the virtual page number, the physical page number, some page status bits like valid and dirty, and an identifier of the owner process. In the case of saving a change in the TLB, we also save the index of the changed TLB entry.

4.2.2 How to Transfer this Information

Any technique that we use to instrument the workloads and dump into the trace buffer the information described above must have two characteristics. First, given the real-time nature of the workloads, it must cause very little perturbation. Second, all the information that we want to pass to the trace must be encoded as memory accesses to addresses that the trace postprocessing program can recognize and decode.

The method that we propose is to encode the information that we want to dump into the trace buffer in the form of memory addresses accessed by dummy references. These accesses we call *escape accesses*. If these accesses are loads, we ignore the value that they read; if they are stores, we make sure that no useful data is overwritten. With this scheme, we can transfer any instrumentation information from the workload to the trace buffer cheaply and non-intrusively with one or more assembly instructions and one or more cache misses. To make the system work, we need to solve three significant problems. These problems are: (1) avoid or handle the address distortion induced by the virtual to physical address translation, (2) prevent the caches from intercepting the escape accesses, and (3) design an efficient and unambiguous encoding of the escape accesses. We consider each problem in turn.

Handling Address Translation

Our escape accesses access addresses that are either TLB-unmapped or have a virtual to physical address translation that we can find out. To start with, escape accesses initiated from within the operating system are usually easy to handle. For example, in both the Silicon Graphics and Alliant systems, the code and static data of the operating system have a fixed virtual to physical mapping. Therefore, we issue escape accesses to virtual addresses in the range used by the operating system code and generate predictable physical addresses to store in the trace buffer. In addition, in the

Silicon Graphics system, if the four most significant bits of the virtual address are set to a certain value, the TLB is bypassed altogether.

The more complicated case occurs if we want to issue escape accesses from an application to its own code or data pages or, in the case of the Alliant system, from the operating system to addresses outside the range assigned to its code or static data. In this case, we select the virtual pages that we will use as target of the escape accesses and inform the trace buffer of their virtual to physical address mapping. We will see an example of how this is done later.

Preventing Cache Hits on Escape Accesses

To prevent the caches from intercepting the escape accesses, we do as follows. In the Alliant system, there is no need for any special support. All data accesses naturally bypass the primary instruction cache and, therefore, are visible to the tracer. Recall that all our escape accesses are data accesses. In the Silicon Graphics system, we rely on the fact that the hardware provides cache bypassing loads and stores. This feature is likely to be common in computer systems. It is provided to allow processors to read memory-mapped registers. In the Silicon Graphics system, any memory access can be made TLB-unmapped and uncachable by setting the four most significant bits of its address to 0xB – bits that are otherwise not used to address memory.

Designing an Efficient Encoding

To show the approach to encoding that we use, we briefly describe a couple of escape accesses from the Silicon Graphics system. More information can be found in [8]. The loads in the escape accesses must be distinguishable from the regular reads in the program. We ensure this by issuing the key escape accesses as byte reads to odd (not even) addresses in a range where only operating system code resides. Obviously, real instruction reads will access even locations only and, therefore, no confusion is possible.

Some of the escape accesses are very simple. For example, *Entering the OS* is encoded as follows:

```
lb r0, 0xB0000001         ;load byte from address 0xB0000001 into register 0
```

This instruction bypasses the TLB and caches to read virtual address 0xB0000001, which gets translated into physical address 0x1. Register 0 is not modified because it always contains the value zero. In practice, we need an extra register to store the constant 0xB0000001. For this reason, we need to save and restore one register.

Other escape accesses, however, are more complex. For instance, when we want to record that a new entry is added to the TLB, we have to send to the trace four pieces of information: the index of the TLB entry changed, the number of the virtual page added, the number of its corresponding physical page plus the page status bits (valid, dirty, etc.), and the identifier of the process that owns the page. To transfer this information, we first read the location that signals *TLB entry change*: 0xB0000005. Then, one at a time, we take each of the four pieces of information to send, store it in a register, shift the register left one bit and set the least significant bit to 1 to make the data odd, set the four most significant bits to 0xB to make the address TLB-unmapped and uncachable, and then byte-read from it. Overall, with 5 escape accesses, we have encoded all this information in the trace.

While the last four addresses read by the escape access sequence can be anywhere in the address space, the postprocessing program will easily identify and interpret each escape in the sequence. Indeed, after finding a *TLB entry change* access in the trace, the trace postprocessing program will look for the next four reads from odd addresses by this processor in the trace. Any other cache misses suffered by this processor while sending the escapes are necessarily instruction misses and hence access even addresses only. Therefore, they cannot be confused with escape accesses.

Finally, to ensure that the sequence of escape accesses appears on the bus as expected, we have to prevent compiler optimizations and interrupts. The optimizing compiler could potentially reorganize the assembly instructions to improve the pipeline. To disable this effect, we surround the assembler instructions that we add with the directives *.set noreorder* and *.set reorder*. This prevents the compiler from making any changes to the sequence of instructions. Similarly, to guarantee that interrupts do not interrupt a sequence of escapes, we disable interrupts while dumping a sequence of escapes. Disabling and enabling interrupts is very cheap: it only requires 2 or 1 assembly instructions to save or restore the status register.

In [8], we give examples of the escape sequences used in the Alliant system. In the Alliant, escape accesses can be loads or stores. In the report, we show the encoding used to identify each of the $47,000$ basic blocks of the operating system. We associate a different escape code to each subroutine and, within a subroutine, a different escape code to each basic block. We then issue an escape access with the appropriate code every time that we enter a subroutine and every time that we enter a basic block. With this, we can record the sequence of basic blocks executed. We only need to set aside one 4-Kbyte page as the target of the escape accesses. A similar approach is done for the basic blocks of applications.

5 Evaluating the Approach

The approach presented offers the possibility of gathering very detailed information of real-time workloads with relatively little perturbation. The actual degree of perturbation depends largely on two factors, namely what events we want to measure and how costly it is to bypass the caches in the machine considered. The first factor determines the number of overhead instructions that need to be executed. For example, if we monitor events that happen with high frequency, the perturbation will tend to be higher. The second factor depends on the specific platform that we are monitoring. For example, in the Silicon Graphics system, the round trip latency of a escape access is about 30 cycles. In the Alliant system, instead, it is about 2 cycles. Of course, the processor does not need to be stalled all this time, but this number is probably proportional to the overheard of the scheme.

Many times we are interested in what data structures and code areas cause most of the misses in the caches of the machine monitored. In this case, it suffices to analyze the accesses that reach the trace buffer. Then, with a few escape accesses, we can identify the virtual addresses of these data structures and code areas.

However, even when many escape accesses are added, this approach induces surprisingly low perturbation. To prove it, we perform an instrumentation-intensive experiment for the Alliant system: we instrument all the basic blocks of the operating system and applications to reconstruct the complete sequence of instructions executed. Since all data accesses are collected by default, this setup allows us to reconstruct the *complete trace of accesses*. In the following, we first present the workloads used for the experiment and then examine the perturbation incurred in the system.

5.1 Workloads Executed

For this experiment, we choose three representative workloads. The first one is *Ccom*. *Ccom* is a set of runs of the second phase of the C compiler, which generates assembly code given the preprocessed C code. Four instances of this program are run on four directories with 11 C files each. Each file is, on average, 49 Kbytes long. The code for this compiler pass consists of about 15,000 lines of C source code. This workload has both system-intensive and compute-intensive phases.

The second workload is *TRFD* [4], a 4-process hand-parallelized version of the TRFD Perfect Club code [6]. The code models the computational aspects of a two-electron integral transformation, and is predominantly composed of matrix multiplies and data interchanges. During execution, the code regularly changes between parallel and serial regimes, while processes synchronize quite frequently. The code consists of about 700 lines of Fortran code and has very large basic blocks. The most important operating system activities in this application are cross-processor interrupts, process synchronization, and gang-scheduling of processes.

The third workload is *ARC2D+Fsck*, a multiprogramming mix with one run of *ARC2D* and two runs of *Fsck* applied to two file systems. *ARC2D* is a 4-process hand-parallelized version of a Perfect Club code [6]. It models a 2-D fluid dynamics code and primarily consists of sparse linear systems and rapid elliptic problem solvers. *Fsck* is a file system consistency check and repair utility. The source code for *ARC2D* contains about 4,000 lines of Fortran code, while the code for *Fsck* contains about 4,500 lines of C code and has moderately small basic blocks. The operating system activity for this workload consists of TRFD-like activity for *ARC2D*, a wide variety of I/O-related code for *Fsck*, and scheduling activity to change between parallel and serial jobs.

The operating system is also a parallel workload in this study since it is instrumented and its address traces are collected.

5.2 Perturbation Incurred

In this section, we analyze the perturbation caused by our approach when we instrument all operating system and application basic blocks. To measure the perturbation, we examine three issues, namely memory overhead, execution speed slowdown, and timing accuracy. We consider each issue in turn. In our experiments, we consider four environments: *NoInst* (neither the operating system nor the applications are instrumented), *OSInst* (the operating system alone is instrumented), *APInst* (the applications alone are instrumented), and *OSAPInst* (both the operating system and the applications are instrumented).

5.2.1 Memory Overhead

One measure of the perturbation caused by the trace generation system is the increase in memory requirements. The increase in memory requirements is caused by the addition of escape accesses to the instructions and the allocation of pages whose only purpose is to be the target of escape accesses. In our system, we add one escape access per routine and one per basic block. The escape instruction takes 6 bytes. The number of pages added to serve as the target of escape accesses is 1 for each application and the operating system. With this scheme, therefore, the average size of the basic block largely determines the relative increase in memory use. In particular, for scientific applications with large basic blocks, the memory overhead will be smaller than for system-intensive workloads where basic blocks are small.

The static sizes of the codes before and after the instrumentation are shown in Table 1. The table is organized with the codes with the highest relative increase in size at the top of the table. As the table shows, code expansion ranges from a mere 2.1% up to 48.3%. Applications with many branches and, therefore, small basic blocks suffer the largest increases. For example, *Ccom* and the kernel increase by 48.3% and 39.1% respectively. On the other hand, regular scientific applications with large basic blocks increase the least. For example, *TRFD* increases by 2.1% only.

Table 1: Static increase in code size due to instrumentation.

Workload	Size		
	Before (Kbytes)	After (Kbytes)	Increase (%)
Ccom	127.0	188.4	48.3
Kernel	930.8	1295.1	39.1
ARC2D	162.9	214.4	31.6
Fsck	69.6	90.1	29.5
TRFD	99.3	101.4	2.1

Overall, the memory overhead is small for a system that can reconstruct the dynamic sequence of instructions. In other instrumentation systems, the expansion induced is much larger. For example, the instrumentation in [10] causes a code expansion of 400%. As a result, our instrumentation does not appreciably change the rate of page faults recorded by the operating system.

5.2.2 Execution Speed Slowdown

A second measure of the perturbation caused by the trace generation system is the slowdown caused in the execution speed of the workloads. The slowdown is caused by the execution of the instructions added to generate the escape references plus any effects that these instructions induce in the miss rate of caches and TLBs and in the paging activity of the operating system. The increase in instructions executed cannot be deduced directly from the data in Table 1, since the table shows static data. Instead, we need to determine the change in dynamic count of instructions.

To measure the execution speed slowdown, we perform timing runs of uninstrumented and instrumented versions of the workloads and kernel. We start with the kernel instrumentation. The effect of the kernel instrumentation is shown in Table 2. The table shows the execution times for the environment without instrumentation (*NoInst*) and with kernel instrumentation only (*OSInst*). For each environment, the table shows the user and system time.

Starting with the *NoInst* environment, we see from Column 3 in the table that these are system-intensive workloads. The system time accounts for a bit less than the user time in two workloads and about 10% of the user time in the other one. Once we move to *OSInst*, the system time increases noticeably (Column 6). This is the result of the extra instructions executed and their effects on the memory hierarchy. The ratio of system times after and before instrumentation is shown in Column 8. From the numbers, we see that the system slowdown is 2.4-3.2. This is a very modest number.

Looking now at the user time (Columns 2 and 5), we see that, as expected, it does not change much for *Ccom* and *TRFD*. However, for *ARC2D+Fsck*, it increases from 62 to 88 seconds. Clearly, instrumenting the kernel affects the uninstrumented application. The reason is likely to be that the code expansion in the new kernel changes the cache mapping of the kernel code and increases the conflicts with the applications. Overall, adding user and system time, the *OSInst* environment is only 1.1-2.1 times slower than *NoInst* (Column 9). This is a small slowdown.

Table 2: User and system execution times for the *NoInst* and *OSInst* environments.

Workload	*NoInst*			*OSInst*			*OSInst/NoInst*	
	User (s)	System (s)	User+ System (s)	User (s)	System (s)	User+ System (s)	System	User+ System
Ccom	47.5	4.5	52.0	47.7	11.0	58.7	2.44	1.13
TRFD	22.9	21.4	44.3	24.6	69.4	94.0	3.24	2.12
ARC2D+Fsck	61.6	47.3	108.9	88.3	126.1	214.4	2.67	1.97

To see the impact of application instrumentation, Table 3 examines an environment with application-only instrumentation (*APInst*) and application plus kernel instrumentation (*OSAPInst*). Comparing the user time for *NoInst* (Column 2 of Table 2) to *APInst* (Column 2 of Table 3), we see that the user time increases to a variable degree due to application instrumentation. Again, the system time stays approximately constant except for the *ARC2D+Fsck* workload where conflicts occur (Column 3 in Tables 2 and 3). Overall, if we instrument applications only, the workloads run only 1.1-1.8 times slower (Column 6 in Table 3).

Finally, Columns 7 and 8 of Table 3 show the user and system time for the *OSAPInst* environment. Comparing these times to the ones with only application or kernel instrumentation, we see that the only cases where the instrumentation causes interference are *ARC2D+Fsck* and the

Table 3: User and system execution times for the *APInst* and *OSAPInst* environments.

Workload	*APInst*			*APInst*/ *NoInst*		*OSAPInst*			*OSAPInst*/ NoInst
	User (s)	System (s)	User+ System (s)	User	User+ System	User (s)	System (s)	User+ System (s)	User+ System
Ccom	68.4	4.7	73.1	1.44	1.41	69.4	11.4	80.8	1.55
TRFD	26.0	20.4	46.4	1.14	1.05	28.8	94.8	123.6	2.79
ARC2D+Fsck	125.6	75.2	200.8	2.04	1.84	137.3	155.7	293.0	2.69

system time in *TRFD*. Overall, as shown in the last column of Table 3, under *OSAPInst*, the workloads run only 1.6-2.8 times slower than under the non-instrumented environment (*NoInst*). This slowdown is substantially smaller than the slowdowns in other address tracing techniques discussed in Section 2. Indeed, some of those techniques involve slowdowns in the order of 10-times for application-only tracing.

5.2.3 Timing Accuracy

Given the real time nature of the operating system, even the 1.6-2.8 slowdown caused by our system could change the timing of critical operating system functions in a way to make our trace simulations inaccurate. For example, the slowdown could cause some operations to become relatively less frequent while some other operations become relatively more frequent. If this effect were significant, then our traces would contain invalid information and could not be used for performance evaluation studies.

To determine if this is the case, we measured the distribution of operating system routines invoked under the *NoInst* and *OSAPInst* environments running the same workload. We then compared the two. If the distributions are significantly different, then the *OSAPInst* traces are invalid; otherwise, we have reasonable confidence that the *OSAPInst* traces model the behavior of the uninstrumented code with enough accuracy.

To get the distribution of operating system routines invoked in the *OSAPInst* environment is easy. Indeed, since we have instrumentation for all the basic blocks of application and operating system, we can determine what routines are being called. Unfortunately, it is not possible to get the same information for the *NoInst* environment. This is because, without escape accesses, the primary cache intercepts an unknown number of references and the call graph cannot be regenerated from the traces. To solve this problem, we add one escape access for each routine. The resulting instrumentation is tiny, namely 1 instruction per routine invoked. Therefore, we claim that such a system is very close to *NoInst*.

Figure 1 shows the results of our measurements for the *OSAPInst* environment and the *NoInst* environment with one escape access per routine. The latter environment we call *LowOSInst*. In the figure, we have numbered the routines according to the order in which they are laid out in memory. The total number of operating system routines is about 2,500. Then, we count the number of times that each routine is invoked under the *LowOSInst* (Chart (a)) and *OSAPInst* (Chart (b)) environments. Finally, we have normalized all the data to 100 invocations. The data in the figure corresponds to the *Ccom* workload; the other two workloads behave similarly.

From the figure, we see that the two environments exhibit similar distributions of routine invocations. Indeed, the height and location of the peaks in the two charts are similar. If the 1.6-2.8 slowdown induced by the instrumentation had perturbed the behavior, some peaks would have become significantly higher than before while others would have become significantly lower. This is not what we see in the plots. It is true that there are some small differences in the peaks marked

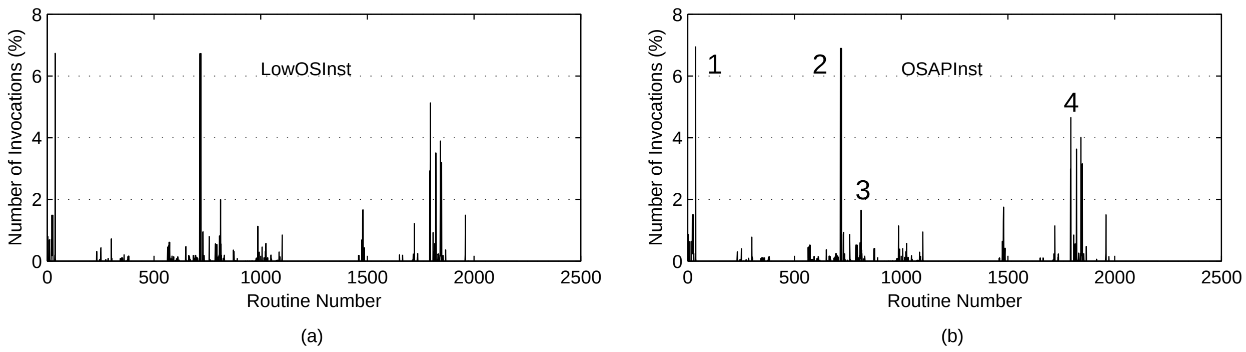

Figure 1: Operating system routines invoked under an environment with a very lightly-instrumented kernel (*LowOSInst*) and one where kernel and applications have their basic blocks instrumented (*OSAPInst*). The routines are numbered in the X-axes according to the order in which they are laid out in memory. The total number of routine invocations is normalized to 100. The data corresponds to the *Ccom* workload.

as 1, 2, 3, 4, which correspond to time-sensitive routines that handle locks and TLB accesses. However, these differences are small and, overall, the charts look remarkably similar. Therefore, we conclude that the two sets of traces are reasonably close to each other and that the traces with the *OSAPInst* instrumentation are accurate.

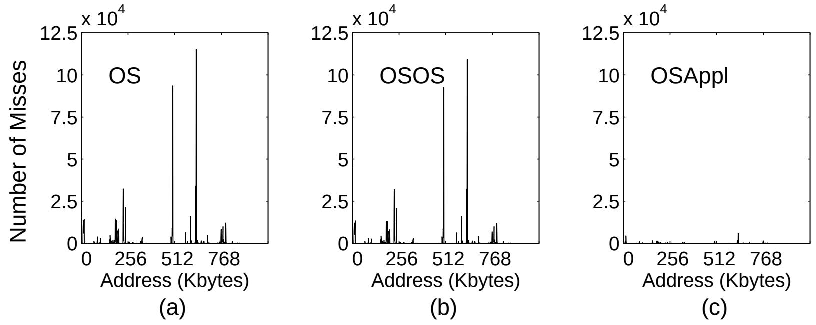

Figure 2: Misses on operating system code as a function of the virtual address of the code.

6 Examples of Uses

We finish this paper by showing two examples of what type of information can be generated with the approach described. Figure 2 shows the number of cache misses on operating system code as a function of the virtual address of the referenced instruction. The data corresponds to 8-Kbyte direct-mapped caches with 32-byte lines, and was generated while running *ARC2D+Fsck*. Chart (a) shows all the misses, which are then decomposed into misses caused by operating system self-interference (Chart (b)) and misses caused by interference with the application (Chart (c)). Misses caused by first-time references are negligible. As a second example, Figure 3 shows the characteristics of the operating system loops that do not call any subroutine. The data corresponds to *Ccom*. The leftmost chart shows a distribution of these loops according to how many iterations are executed per loop invocation, while the rightmost chart shows a distribution of these loops according to their static size. Clearly, many different types of information can be gathered.

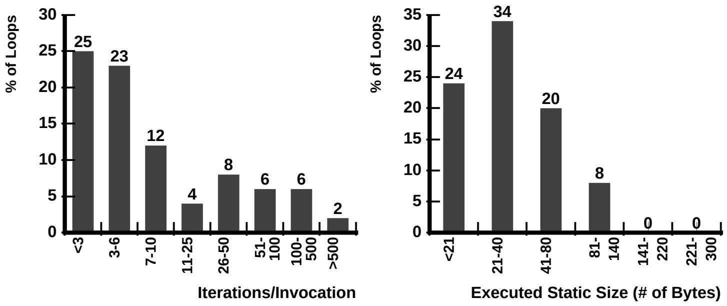

Figure 3: Characteristics of the operating system loops that do not call any subroutine.

7 Conclusion

Address trace analysis is a popular method to evaluate the memory system of computers. However, obtaining accurate traces of operating system, multiprogrammed, and parallel workloads in multiprocessors is hard. The real-time nature of these loads usually requires trace-gathering hardware performance monitors. Unfortunately, such devices tend to have many limitations: they often only capture physical addresses, can only collect references that miss in on-chip caches, and can only monitor a very small time window.

In this paper, we improved the capability of such systems. We introduced a simple methodology to instrument operating system and applications in a multiprocessor to transfer a wide variety of information to the trace-gathering hardware with little perturbation. For example, we transfer the virtual to physical mapping of the addresses accessed or the sequence of basic blocks executed. Each piece of information was transferred as cheaply and non-intrusively as one or very few machine instructions and one or very few cache misses. Using this approach, we instrumented every basic block in the operating system and applications to reconstruct the whole sequence of instructions executed. The resulting workload slowed down by only 1.6-2.8 times. This is in contrast to the 10-time slowdown reported for software-based instrumentation of applications only. This modest slowdown caused by the instrumentation leaves the behavior of the operating system largely unaffected.

8 Acknowledgments

We thank Pen Yew, Wen-Mei Hwu, John Hennessy, Anoop Gupta, the referees, and the graduate students in our group for their feedback. We also thank Tom Murphy and Perry Emrath for their help with the experimental system. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] C. Alexander, W. Keshlear, F. Cooper, and F. Briggs.
Cache Memory Performance in a UNIX Environment.
In *Computer Architecture News*, pages 41–70, June 1986.

[2] Alliant Computer Systems Corporation.
FX/Series Product Summary.
Littleton, MA, 1986.

[3] J. B. Andrews.
A Hardware Tracing Facility for a Multiprocessing Supercomputer.
Technical Report 1009, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, May 1990.

[4] J. B. Andrews.
Parallelization of TRFD.

Internal Document, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, November 1991.

[5] F. Baskett, T. Jermoluk, and D. Solomon.
The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second.
In *Proceedings of the 33rd IEEE Computer Society International Conference - COMPCON 88*, pages 468–471, February 1988.

[6] M. Berry et al.
The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers.
International Journal of Supercomputer Applications, 3(3):5–40, Fall 1989.

[7] B. Chen.
Software Methods for System Address Tracing.
The 4th Workshop on Workstation Operating Systems. Napa, California, 1993.

[8] R. Daigle, C. Xia, and J. Torrellas.
Low Perturbation Address Trace Collection for Operating System, Multiprogrammed, and Parallel Workloads in Multiprocessors.
Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign, March 1996.

[9] H. Davis, S. Goldschmidt, and J. Hennessy.
Multiprocessing Simulation and Tracing Using Tango.
In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, pages 99–107, August 1991.

[10] S. Eggers, D. Keppel, E. Koldinger, and H. Levy.
Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor.
In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 37–47, May 1990.

[11] P. A. Emrath, M. S. Anderson, R. R. Barton, and R. E. McGrath.
The Xylem Operating System.
In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, pages 67–70, August 1991.

[12] J. Larus and E. Schnarr.
EEL: Machine-Independent Executable Editing.
In *In Proceedings of SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[13] J. Mogul and A. Borg.
The Effect of Context Switches on Cache Performance.
In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, April 1991.

[14] D. Nagle, R. Uhlig, and T. Mudge.
Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architecture.
CSE-TR147-92, University of Michigan, May 1990.

[15] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta.
Fast and Accurate Multiprocessor Simulation: The SimOS Approach.
In *IEEE Parallel and Distributed Technology*, volume 3, Fall 1995.

[16] A. Srivastava and A. Eustace.
ATOM: A System for Building Customized Program Analysis Tools.
In *In Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, June 1994.

[17] M. Talluri and M. Hill.
Surpassing the TLB Performance of Superpages with Less Operating System Support.
In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, October 1994.

[18] J. Torrellas, A. Gupta, and J. Hennessy.
Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System.
In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.

[19] J. Torrellas, C. Xia, and R. Daigle.
Optimizing Instruction Cache Performance for Operating System Intensive Workloads.
In *Proceedings of the 1st International Symposium on High-Performance Computer Architecture*, pages 360–369, January 1995.

[20] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest.
Trap-Driven Simulation with Tapeworm II.
In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–47, October 1994.