SEPTEMBER 1994

WRL Research Report 94/6

Software Methods for System Address Tracing: Implementation and Validation

J. Bradley Chen
David W. Wall
Anita Borg

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Software Methods for System Address Tracing: Implementation and Validation

J. Bradley Chen
David W. Wall
Anita Borg

September 1994

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

Systems for recording address traces of operating system activity have frequently relied on special-purpose hardware and microcode modifications for data collection [1, 2, 10, 11, 30, 32]. In the last decade, changes in computer systems design have made the implementation of such hardware and microcode-based tracing systems impractical. This paper documents the evolution of a group of software methods to collect system traces. The tools require no special-purpose hardware and no hardware modifications. We have applied these tools to three substantially different operating systems and two processor architectures. This paper describes the instrumentation techniques, the means used to assure the quality of the collected data, and our evaluation of correctness and accuracy of traces. Our experience shows that software methods can yield trace of very good quality, and can be used to measure complex software systems.

1. Introduction

Address tracing is an important technique for measuring the dynamic behavior of computer software systems and the interactions between software and hardware on a computer system. The two approaches to collecting address trace data (which we will sometimes call simply *trace*) are hardware methods and software instrumentation.

Hardware methods involve the use of modified microcode or a special-purpose device to intercept and record memory references as they occur. With software methods, the program or programs of interest are augmented with instrumentation code, such that address trace data is generated as a side-effect of program execution. This suggests three problems with software methods:

- The instrumentation process is intrusive and can change the behavior of the traced system in a substantial way.
- Trace from different address-spaces tends to be buffered independently and hence partitioned on a per-address-space basis.
- Operating system code is difficult to instrument for address tracing.

Hardware methods avoid the problems of software instrumentation by intercepting trace information at a very low level, such that all activity is captured indiscriminate of source, and the behavior of software is unaffected. Unfortunately, several properties of current processor design make hardware-based tracing difficult:

- Microcode-based designs are not useful because current processors do not use reloadable microcode.
- Current processors tend to incorporate memory system components such as caches and translation buffers into an integrated microprocessor package, such that the required address trace information is transferred on submicron-sized structures sealed inside the computer chip. This makes them impractical to access with a hardware monitor.
- Modern high-performance computers operate at very high clock speeds. Consequently, any monitoring device must also operate at a very high speed. This makes such a device difficult and expensive to build.

The problems with hardware tracing led us to take a harder look at software-based methods. In this paper we describe and discuss the three tracing systems we have implemented. All are variants of the Unix operating system.

- Traced Tunix: Tunix was derived from DEC Ultrix, version 4.1, and ran on the DECWRL Titan [5].
- Traced Ultrix: A traced version of Ultrix version 4.2 for the DECstation 5000/200.
- Traced Mach 3.0: A traced version of Carnegie Mellon's Mach 3.0 microkernel (MK78) and UNIX server (UX39) for the DECstation 5000/200.

The remainder of the paper is structured as follows. After discussing previous work, Section 3 discusses the design of the tracing system, beginning with instrumentation tools and fundamental notions of system design, then going on to describe details on the Tunix, Ultrix, and Mach 3.0 tracing systems and how they differ. Section 4 discusses sources of distortion in software-based tracing systems and how we avoided them. Section 5 discusses measurements we made to estab-

lish that the trace data was a reasonable indicator of real system behavior. In the last section we briefly summarize our conclusions.

2. Previous Work

Software methods have been applied extensively to study user-only traces, yielding results in cache behavior [15, 16, 17, 26, 28], prefetching [6], the importance of long traces [5], the impact of context switches [20], and studies of TLB and page behavior [9, 18, 29]. These user-only studies are useful but limited, as system activity can have a large impact on overall performance [2, 12, 30]. More recent work documenting significant performance problems for system execution on RISC-based computer systems [3, 24] suggests that system behavior needs more attention in performance studies and hardware design.

Clark and Emer were among the first to emphasize the importance of system activity when modeling memory system behavior. They used direct measurements of hardware to study cache performance in the VAX 11/780 [10] and to evaluate the VAX 11/780 translation buffer [12]. They point out that direct measurement and simulation have complementary advantages and disadvantages. They also recognized the problem of distortion of system behavior due to tracing.

A review of more recent hardware tracing projects demonstrates how the obstacles of hardware methods limit their applicability in current research.

In the ATUM system, the microcode of a DEC VAX 8200 was modified to record an address trace of system and user execution [2]. The method was applied for both VMS and Ultrix systems to test a variety of cache configurations. ATUM has several drawbacks. The foremost is that it can only be used on microcoded processors. Also, long contiguous trace was not possible. The ATUM designers proposed *trace stitching* to address this limitation.

Two interesting projects used hardware monitors to collect system traces of multiprocessor workloads. Researchers at Carnegie Mellon University traced an Encore Multimax computer [32]. Researchers at Stanford traced cache misses only [30]. Cache misses are less frequent than memory references, and this eases the requirements of the hardware device.

Two recent hardware projects used address traces of uniprocessors. The Monster system [21] uses a logic analyzer to capture signals from the CPU chip of a DECstation 3100. They have applied their system to study TLB behavior [22] and the system issues for memory system design [23]. The BACH system has been used to collect address trace on both Intel 80486 and Motorola 68030 based machines [14]. To date they have not demonstrated the applicability of their system to RISC based computers.

In our work we used two instrumentation tools, Mahler [34] and epoxie [35]. There are numerous other software instrumentation tools [4, 13, 27], but none to our knowledge have been used to collect address traces of operating system behavior.

3. The WRL-CMU Tracing Systems

3.1. High-Level Design

The design of the tracing systems was motivated by the need for accurate simulations of the large memory systems that are required by state-of-the-art processors. This had the following implications:

- The traces must be complete. They must represent the kernel and multiple users as they execute on a real machine. The memory references must be interleaved as they are during execution rather than being artificially interleaved separate traces.
- Traces must be accurate. The mechanism used must not distort execution to the extent that the behavior of the system is no longer realistic.
- Traces must be flexible. It must be possible to pick and choose the processes to be traced, optionally trace kernel execution, and turn tracing on and off at any time.
- The traces must be long enough to make possible the realistic simulation of very large caches. Since traces of the required length can outstrip storage capacity, trace analysis that must be done off-line against stored traces is unacceptable.

Figure 1 shows a high-level diagram of the tracing system. The system involves three kinds of entities: traced user processes, the traced kernel, and an analysis program which consumes the trace. The kernel controls the tracing system. Appropriate mechanisms are used to avoid tracing kernel activity that occurs on behalf of the tracing system.

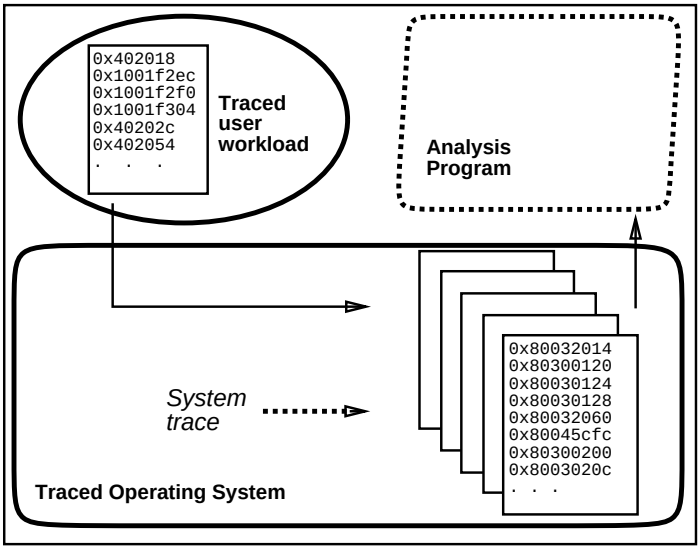

Figure 1: Overview of the tracing system.

At any instant during a tracing experiment the system is operating in one of two modes: trace-generation or trace-analysis. During trace-generation, trace from user-processes goes first into a per-process buffer. When that buffer becomes full, a kernel trap occurs and the per-process trace is copied into the large in-kernel buffer. When the in-kernel buffer becomes full, the system switches from trace-generation to trace-analysis, during which an *analysis program* (such as a memory system simulator) digests the trace. Analysis continues until all pending trace has been analyzed and the in-kernel buffer is empty.

In addition to copying trace from per-process buffers when they become full, available trace is copied into the kernel each time the kernel is activated. As the kernel is invoked between every change of context, the interleaving of trace from all the various sources is preserved.

A certain number of kernel modifications were required to support user tracing, independent of the generation of kernel trace. An in-kernel trace buffer was set up. This is allocated statically at boot time and is never seen by the kernel memory management subsystem. Exception handlers were modified to copy trace from per-process buffers into the in-kernel buffer whenever traced user processes are interrupted. Further, a mechanism is provided for the analysis program to extract trace from the in-kernel buffer. In Tunix and Ultrix, a memory special file is used (similar to `/dev/kmem`). In Mach 3.0, the in-kernel buffer is mapped into the virtual address space of the analysis program.

A kernel call was added to both systems, to provide a mechanism for user-level analysis programs to control tracing. Process creation was modified to initialize tracing data structures. Scheduler modifications were used to insure that traced processes are inactive during trace analysis.

3.2. Software Instrumentation

This section describes the software instrumentation performed by the CMU version of a tool called *epoxie* [35], which was used for the traced Ultrix and Mach 3.0 systems on the DECstation. The tools used for the traced Tunix system differ in several ways; in particular they were integrated into the compiler/loader system, but the overall approach is similar to that used by epoxie.

Epoxie is similar in spirit to the *pixie* tool from MIPS Computer Systems [27], which can be used to insert address-tracing code into an executable. Since the instrumentation code causes the program text to expand considerably, addresses of procedures and branch targets change; address references must be adjusted in the instrumented version if the program is to run correctly. Pixie does some of this address correction statically, when the original executable is rewritten as an instrumented executable, but it must do part of it dynamically, by including a complete address translation table in the instrumented executable and doing lookups in this table during execution of the instrumented program.

Epoxie differs from pixie in that it rewrites object files at link time. Modifying object code at link time is easier than modifying an executable, because the symbol and relocation tables present in object code allow epoxie to distinguish unambiguously between uses of addresses and uses of coincidentally similar constants. This information also allows all address correction to be done statically, incurring no runtime overhead. The addition of address-tracing code results in a significant increase in text segment size. For this project, extensive modifications were made to epoxie to minimize text expansion. The text growth factor ranges between 1.9 and 2.3*. This compares very favorably with pixie, QPT [4], and the original epoxie, all of which expand the

*Actual growth depends on the length of basic blocks and the density of memory instructions.

text by a factor of 4-6 when used for address tracing[**]. It should be noted that minimal text growth was not a design objective for any of the earlier tools.

Note that the expansion of traced text does not affect the trace addresses generated, as the addresses seen by the simulator corresponding to the uninstrumented binary. The motivation for this modification was to minimize the additional I/O and VM behavior that occurs as a result of text growth. I/O and VM effects are discussed in greater detail in Section 5.

A last crucial difference between epoxie and other instrumentation tools is that the other tools work only for single application programs. Epoxie has the flexibility to be used in a tracing system with multi-process workloads, threaded tasks in Mach 3.0, and operating system kernels.

Epoxie inserts trace-collecting code at the beginning of each basic block and before every memory instruction of the original program text. Figure 2 shows an example of a code sequence before and after instrumentation.

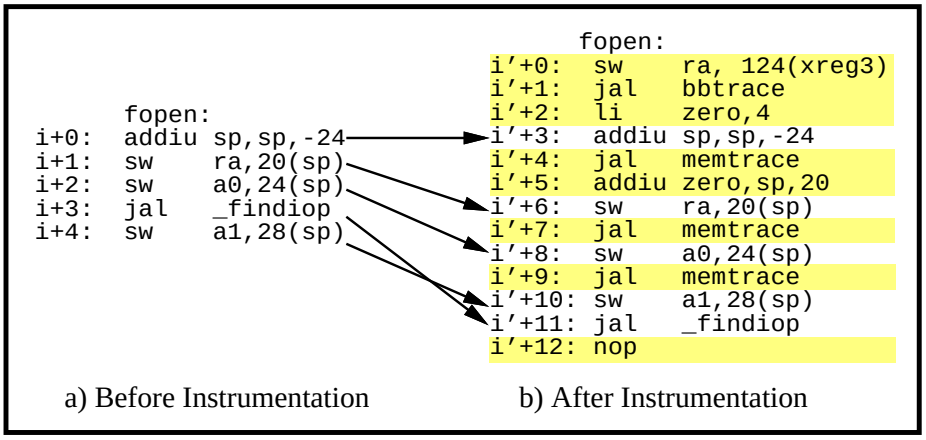

Figure 2: Instrumentation by epoxie

Each basic block is preceded by a three instruction sequence, as in instruction `i'+0 . . i'+2`. The jump instruction at `i'+1` is a call to a basic block trace routine `bbtrace` that will store the `jal`'s return address into the trace buffer. During trace analysis, the trace parsing library will use static information about the binary image to map this address to the correct basic block address in the original (uninstrumented) binary.

The `jal` instruction destroys the return address register `ra`, so instruction `i'+0` saves `ra` in the trace bookkeeping area before `bbtrace` is called. `bbtrace` and `memtrace` restore the contents of `ra` before they return. The delay slot of the `jal bbtrace` contains a special no-op (instruction `i'+2`), a load-immediate to the read-only register `zero`, with the number of words of trace generated by the basic block in the immediate field. This will be used by `bbtrace` to determine if there is enough room in the user trace buffer for trace from this basic block to be stored.

[**]For a `gcc` binary with 688128 bytes of text, `pixie -t gcc` grows program text to 4131968 bytes. `Epoxie -t gcc` grows text to 3780608 bytes. QPT expands *gcc* text by a factor of 5.5 [19]. The modified epoxie grows text to 1515520 bytes.

The tracing system requires three registers for its own use, referred to symbolically as `xreg1`, `xreg2`, and `xreg3`. Uses in the original binary of these *stolen registers* are replaced with sequences of instructions that use a ‘‘shadow’’ value for the register, in memory.

Memory instructions are typically expanded into a two instruction sequence, a `jal memtrace` with the memory instruction in the delay slot, as with `i+2` from Figure 2. `Memtrace` partially decodes the instruction in the branch delay slot to compute the address of the memory reference. Certain hazard conditions sometimes make it impossible to put the actually memory instruction in the branch delay slot. An example is instruction `i+1` in the example above, which reads the `ra` register. For such cases, a no-op with the same base register and offset as the memory instruction is used in the delay slot, and the real memory instruction is issued after the call to `memtrace`.

For a given input, a traced program executes many more instructions than the original binary, so execution time is longer. In Section 4 we discuss sources of distortion in the traced system and how we have controlled them.

3.3. Tracing the Kernel

Features to facilitate complete system tracing are important in our design. An example is the trace format. In the Ultrix and Mach 3.0 systems, a trace entry for a basic block or memory reference is a single machine word. This means that a single machine instruction records a complete trace entry. In this way, trace entries remain contiguous, with no locks or other protection mechanisms required. Another feature that helps accommodate system tracing is that control of the tracing system resides in the kernel. This centralized control makes it possible to preserve the interleaving of trace from various sources. In systems like Pixie and QPT where trace is managed at user level, preserving this interleaving is difficult.

Several peculiarities of operating system kernels make instrumentation a substantially different problem from instrumenting an application program. The foremost is the presence of uninstrumented code. Certain parts of the kernel are not rewritten by the instrumentation tool, either because they are part of the tracing system and should not be traced, or because they are too delicate to be rewritten mechanically. Uninstrumented code in the traced kernel must be carefully handled so as to preserve and maintain the state of the tracing system.

A second problem with tracing the kernel is the need to manage the tracing system. Traced applications are serviced by the trace-control subsystem when their trace buffers become full, with per-process user trace copied into the large in-kernel buffer. Trace of operating system activity goes directly into the in-kernel buffer, so the in-kernel buffer can become full at an arbitrary point during system activity. However, servicing the full buffer is a complicated operation, and cannot be scheduled arbitrarily. Provisions must be made for critical system operations to complete before tracing is suspended.

A third problem is the concurrency introduced by interrupts and exceptions. With traced user activity, activity from concurrent traced user-level activities is always isolated by an invocation of the kernel. This provides an occasion for the kernel to maintain trace system state. There is no such opportunity for a traced kernel, as no intermediate party is available to maintain the kernel’s tracing state when the kernel itself is interrupted by an exception. To address this

problem, the exception handling mechanism in the kernel must be modified to correctly handle trace state, and the trace-analysis system must correctly handle situations when arbitrary kernel activity is interrupted by an exception.

All relevant parts of the kernel are traced. Routines too delicate to be instrumented by epoxie were instrumented by hand. Certain code, executed only at boot time or after an unrecoverable system error, is not instrumented.

3.4. Tunix on the Titan

Our first tracing system was implemented for the Titan [5], an early experimental RISC workstation with a 45ns cycle time and different register sets for user and kernel. The Titan ran a modified version of Ultrix called Tunix. Process and memory management functionality were rewritten for Tunix.

All Titan compilers used a common intermediate language, Mahler, which defined a Mahler abstract machine. The Mahler implementation [33, 34] consists of a translator and an extended linker. Object modules produced by Mahler contain sufficient supplementary information to support the code modification required for address trace generation. In particular, basic blocks and their sizes are identifiable at link time. The linker augmented code to be traced with instrumentation code so that traced programs record addresses and lengths of basic block and load and store target addresses when executed.

A single large trace buffer was managed by the operating system and mapped into every address space. In traced workloads, the compiler reserved five of the 64 user registers for use by the tracing system. Since all tracing data was sent to the same buffer, trace data was correctly interleaved for multiple processes and the kernel.

Only specially linked programs were traced. Kernel tracing was turned off when the kernel acted on behalf of an untraced program. In particular, it was turned off while the trace analysis program was running.

Tunix kernel tracing worked well enough to demonstrate that software-based kernel tracing was possible. Preliminary cache simulation experiments showed that kernel cycles per instruction (CPI) were three times user CPI, and had a significant effect on overall CPI.

Unfortunately, there were a number of problems. As kernel address references were to physical addresses and user references were to virtual addresses, it was difficult to determine when a user and a kernel address actually referred to the same memory location. Also, portions of the Tunix kernel, such as the software TLB miss handler, were not traced. Finally, the internals of Tunix were sufficiently different from commercial operating systems that we were reluctant to draw general conclusions from the behavior of Tunix. The traced Tunix system established the potential of software-based system tracing, and gave us the necessary experience to move our techniques to a more mainstream operating system. The Tunix tracing system also produced a collection of single and multi-task user-level traces on tape, which were made available to the community for use in memory system research.

3.5. Ultrix on the DECstation 5000/200

Epoxie was used to instrument both Ultrix and Mach 3.0. With the Titan machine architecture and with the instrumentation tools integrated with the compiler system, it was straightforward in Tunix to reserve registers for address tracing. In contrast, epoxie operates on binaries *after* compilation, so registers reserved for tracing had to be ‘‘stolen.’’ The necessity of register-stealing complicated the implementation of the tracing system, creating additional trace-system state to be maintained and additional invariants to be observed.

Another source of added complexity in the DECstation tracing systems was the handling of nested interrupts. The Titan had only one interrupt level, so nested interrupts were impossible. The nested interrupts on the DECstation require the tracing system to use a stack to maintain its state during multiple nested system invocations.

An interesting change between traced Tunix and traced Ultrix is the handling of basic block records. Mahler and Epoxie both generate static information describing each basic block (number of instructions, position of loads and stores). This information is used when the trace is analyzed, to determine the correct interleaving of instruction and data memory references. In traced Tunix, the basic block records were written into the trace along with the traced addresses. In the Ultrix system, only the basic block address is written. A lookup table is used in the trace parsing library to find static information for a given basic block address. One advantage of this technique is it makes the trace more concise, so the trace takes less space and less time to write. Another advantage is that the basic block lookup creates an opportunity for implementing special behaviors for a specific basic block address. An example is hand-traced code. The trace-parsing system can recognize the basic block record of a hand-traced routine as special, and respond accordingly. Another example is instruction counting, with flags in basic block records to start and stop counters. An example application of these counters is measuring activity in the idle-loop.

3.6. Mach 3.0 on the DECstation 5000/200

Mach 3.0 is a microkernel that implements and exports a small number of low-level system services, with higher-level services implemented in a user-level UNIX server. The Mach 3.0 virtual memory interface [25] permitted a number of improvements in the implementation of the tracing system. In the analysis program, trace was extracted from the kernel by mapping the in-kernel buffer into the analysis programs address space, eliminating copying and buffering of trace data.

Another use of Mach 3.0 virtual memory primitives is dynamic allocation of the per-process trace pages. In the Ultrix system, a flag was set in the executable image to indicate that a process was traced. This flag is checked when a traced program is started. Traced programs get per-process trace pages, and are scheduled according to the state of the tracing system. The Mach 3.0 system identifies traced programs by detecting references to the per-process trace pages. This feature in Mach 3.0 is particularly important for the implementation of multiple traced threads in a single address space, as independent trace pages are allocated for each thread. Context-switching code in the kernel maps the correct per-thread pages when a new thread is activated.

4. Maintaining Trace Quality

Instrumenting the system involves substantial modifications to every instruction of active system code. As such, one of the original goals in the design of the tracing system was to control the impact of instrumentation on system behavior.

4.1. Avoiding Trace Distortion

Tracing with software methods induces two kinds of distortion on system behavior, *memory dilation* and *time dilation*.

Memory Dilation

A program instrumented with epoxie is about a factor of two larger than its untraced counterpart. This can affect paging and TLB miss behavior. We avoid perturbations due to paging behavior by collecting our traces on a machine with a large physical memory, such that pageouts do not occur. We feel this is a reasonable simplification, since the main utility of our tools is in analyzing memory system performance, and most aspects of memory system performance become irrelevant when significant paging activity is present.

TLB behavior is slightly more subtle. The DECstation address spaces is divided into four segments, two mapped and two unmapped. All kernel text and most kernel data is referenced through the unmapped segments; hence these references do not affect the TLB. The two mapped segments do require translations from the TLB, and each handles TLB misses differently. A miss to the *user segment* is called a *UTLB* miss and is handled in software via a dedicated exception vector and a nine-instruction miss handler routine. A miss to the *mapped kernel segment* is called a *KTLB* miss. They are handled through the general exception mechanism, which is much slower (several hundred instructions). Fortunately, KTLB misses are more rare.

Instrumentation causes the number of user text pages to grow by a factor of two. With twice as many text pages, UTLB miss behavior can differ substantially between traced and untraced workloads. Because of the different behavior, trace from the actual user TLB miss handler would not be representative of the untraced system. Rather than tracing the UTLB miss handler, we simulate the TLB, and use misses in the simulator to synthesize the activity of the UTLB miss handler.

Mapped kernel memory is used primarily to map page table pages. If instrumentation changed the number of page table pages required to map user text, then KTLB miss behavior could be affected. Fortunately, each page table page can map 4 megabytes of contiguous memory. As the largest traced binary has less than 2 megabytes of text, the number of page table pages it requires does not change, and the behavior of the KTLB miss handler is unaffected.

Time Dilation

The instructions added by software instrumentation cause traced programs to execute about fifteen times more slowly than their untraced counterparts. Temporal relationships for activity that depends on the speed of CPU instruction execution are unaffected, as the slowdown for all instrumented code is roughly the same. Time dilation occurs because activities independent of CPU speed appear to occur about fifteen times faster for the traced system. For the workloads we have considered, this affects clock interrupts and the latency of I/O operations. Adjusting for

clock interrupts was straightforward: we configured the system clock to interrupt at 1/15th the standard rate.

We have not modified I/O behavior to account for time dilation, as this would require subtle system changes that might themselves introduce other distortions. Instead, we estimate I/O delays using a count of the number of instructions executed while waiting in the system idle-loop. We estimate I/O delays in the untraced system by multiplying idle time in the traced system by a scaling factor of fifteen. The approximation this gives is very rough, but largely adequate for our purposes, as I/O delays are of little interest in memory system behavior.

Scheduler policy is also affected by time dilation, but is an issue we have chosen not to address. Instead, we concentrate on workloads such as single process workloads and client-server systems. For these workloads, all context switches are determined by client-server relationships, and scheduler policy is irrelevant. For accurate traces of timesharing workloads, it would be necessary to scale I/O delays and adjust scheduler policy to replicate untraced behavior. It should be possible to improve the behavior of the traced system, although perfect reproduction of traced behavior is not a practical goal. Given current trends toward single-user machines and away from timesharing, the limitation of client-server systems leaves ample domain for our research.

4.2. Page Mapping Policy

The virtual to physical page map is determined by policy implemented in operating system, and can have significant impact on memory system behavior. [7, 18] An address trace obtained through software methods contains *virtual* addresses, yet caches are often indexed by *physical* addresses. A trace-based simulation of such a physical cache requires some virtual-to-physical address translation. The most straightforward approach is to implement the desired page mapping policy in the simulator. The traced Ultrix and Mach 3.0 kernels also provide the option of extracting the page-map from the running system.

4.3. Defensive Tracing

When possible, the validity of tools was tested in isolation from the rest of the system. The system was further tuned and corrected by looking for anomalies in measured behavior. In this section we discuss redundancy and error modes built into the tracing system that are helpful for avoiding certain kinds of errors. In the next section we discuss our final means of evaluating the quality of trace, measuring the ability of a trace driven simulation to predict measurements of an uninstrumented system.

The following discussion applies primarily to the traced Ultrix and Mach 3.0 systems.

The correctness of trace generated by epoxie instrumentation was validated by comparing epoxie trace for deterministic user programs to trace from a CPU simulator. The verification of trace from epoxie against trace from an independently developed CPU simulator establishes with a high degree of certainty the correctness of epoxie instrumentation.

In the operating system kernel, code rewritten by epoxie co-exists with hand-instrumented code, as well as the uninstrumented code that implements certain parts of the tracing system.

Unlike user code, the kernel is significantly involved in controlling the state of the tracing system. A number of approaches were used to ascertain that tracing the kernel did not introduce errors or unexpected trace distortion.

- The format of trace contains a significant degree of redundancy, such that missing words of trace or erroneous writes into the trace are detected with a very high probability. Conditions checked include (i) that each instruction basic block address is valid for the address space in question, and (ii) that in each basic block the expected number of memory operations occurs.
- A large number of sanity checks were used to verify that trace was not being misinterpreted. For example, the simulator checks that all kernel instruction addresses are in the kernel instruction address space.
- Reference counting tools were used to make a dynamic count of the number of times each instruction in the kernel was executed. In this way it was possible to identify anomalous system activity caused by errors in the tracing system.

Each time the tracing system changes from trace-generation mode to trace-analysis mode, a certain amount of ‘‘dirt’’ is introduced into the trace, where the activity reflected in the trace does not accurately reflect a run of an uninstrumented system. No user-level trace is lost, but it is possible for some amount of errant system activity to be measured or ignored. As an example, an I/O request might be made during trace-generation mode, but complete during trace-analysis mode. The trace from the completion of the I/O request would then be lost. The approach taken to minimize the inaccuracies introduced by these transitions was to be sure they are rare, by making the in-kernel trace buffer large. The current system uses a 64 megabyte buffer. A buffer of this size permits approximately 32 million instructions of continuous execution between trace analysis phases. For an untraced system, this corresponds about two seconds of continuous execution.

4.4. Truths Revealed

In debugging the tracing system, a procedure used repeatedly was to identify anomalous behavior indicated by simulator output and trace it back to a bug in the simulator or simulation model. Eventually, these investigations stopped revealing problems in the experimental system, and began to expose unexpected behavior from the actual hardware and operating system implementation. Some of these behaviors include:

- A bug in the instruction cache flushing routine caused an excessive number of uncached instruction references in Mach 3.0.
- System activity accounts for about 1% of execution time in *tomcatv*, but system policy in the virtual-to-physical page selection can cause execution time to vary by over 10%.
- Conservative write policies in Ultrix induces greatly increased I/O delays.

Overall, the ability of the tracing system to reveal these unexpected behaviors demonstrates that the combined tracing/simulation system accurately reflects the behavior of the uninstrumented system.

5. Validation of Methods

This section describes how we measured the fidelity of the experimental system to real system behavior. The measurements use the example workloads are described in Table 1. Overall, these measurements demonstrate that the tracing/simulation system is a good model for real system behavior.

Workload	Description
sed	The UNIX stream editor run three times over the same 17K input file.
egrep	The UNIX pattern search program run three times over a 27K input file.
yacc	The LR(1) parser-generator run on an 11K grammar.
gcc	The GNU C compiler (gcc) translating a 17K (preprocessed) source file into optimized Sun-3 assembly code.
compress	Data compression using Lempel-Ziv encoding. A 100K file is compressed then uncompressed.
espresso	A program that minimizes boolean function run on a 30K input file.
lisp	The 8-queens problem solved in LISP.
eqntott	A program that converts boolean equations to truth tables using a 1390 byte input file.
fpppp	A program that does quantum chemistry analysis. This program is written in Fortran.
doduc	Monte-Carlo simulation of the time evolution of a nuclear reactor component described by 8K input file. This program is written in Fortran.
liv	The Livermore Loops benchmark.
tomcatv	A program that generates a vectorized mesh. This program is written in Fortran.

Table 1: Experimental workloads with execution times for a DECStation 5000/200.

Except where indicated, all programs are written in C. The bottom four workloads are floating-point intensive.

5.1. Program Execution Time

We used a high resolution timer to measure execution times of the workloads. In Table 2, we compare the measured times with times predicted from a trace-driven simulation of the DECstation 5000/200 memory system.

The predicted times in Figure 2 include contributions from four different sources:

- CPU cycles
- memory system stalls
- arithmetic stalls
- I/O stalls

Each instruction executed contributes one CPU cycle to the total execution time. Memory system stall cycles are calculated by multiplying counts of penalty events (cache read misses, uncached reads, and write-buffer stalls) by the number of stall cycles per event. Pixie [27] was used to estimate arithmetic stalls, as the tracing system does not measure these events.

The estimate of I/O stalls is derived from a count of idle-loop instruction references made from the memory reference trace. Information on idle-loop activity from the trace must be ad-

	Mach 3.0		Ultrix	
workload	measured	predicted	measured	predicted
sed	0.58	0.48	0.48	0.54
egrep	2.05	2.02	1.94	1.90
yacc	1.70	1.68	1.80	1.79
gcc	2.26	3.21	4.10	4.16
compress	1.38	1.17	1.26	1.11
espresso	6.03	6.21	6.43	6.40
lisp	62.0	56.6	53.3	53.6
eqntott	66.1	65.7	65.6	65.8
fpppp	15.9	16.7	15.9	15.7
doduc	20.7	21.4	21.7	21.2
liv	1.29	1.29	1.17	1.26
tomcatv	139.2	137.0	155.4	153.5

Table 2: Run Times, measured and predicted, in seconds.

The predicted times are the sum of machine cycles from four different sources: instruction execution, memory system stalls, I/O time, and arithmetic stalls. The first three values are measured by the tracing system. Estimates for arithmetic stalls are as measured by pixie [27]. Execution times in this table are for runs with revision 3.0 of the floating point unit. Also, the buffer cache was warmed with executable text before program execution.

justed to compensate for the effects of time-dilation on the execution of idle loop. As an example, consider a workload that executes 15 million instructions in the idle loop while waiting for completion of synchronous disk I/O. This corresponds to some amount of real time required for I/O operations by the disk. Address tracing does not change the latency of disk operations, but time dilation changes the execution rate of the idle-loop. Suppose that instrumented code is slower than uninstrumented code by a factor of fifteen. Then only 1/15th as many or 1 million idle-loop instructions will be recorded in the trace. Idle-loop instruction counts from the trace must be scaled to compensate for the slowdown in idle-loop execution. For the predictions of program execution time from trace data, 15 is used as an estimate of the effect of instrumentation on the idle loop.

The running times from simulator data are rough estimates, and are subject to error from a number of sources.

- *Disk Latency and Idle time.* The simulator's model of disk delays is only an approximation of real behavior. This approximation introduces distortions to time estimates in two ways. First, Some system activity is missed when tracing is interrupted during a disk request. Second, tracing changes the behavior of disk read ahead. Some read-ahead requests which complete in the traced system don't complete in the standard system. This results in idle time for the standard system that does not occur in the traced system.
- *Lack of pipeline model.* The simulation system does not model the CPU pipeline. Although there is a mechanism to model the correct sequencing of instruction reads and data reads and writes, two other behaviors are not modeled:
  - Floating point latency can overlap with write buffer cycles and cache misses in the DECstation 5000/200. This overlapping is not modeled in the simulator.
  - The simulator does not account for cycles required to enter and exit exception handlers.

- *Page mapping policy*. Cache performance can vary significantly depending on the virtual to physical page mapping in use. This affects the repeatability of workload behavior, particularly for the random page mapping policy used in Mach 3.0.
- Clock interrupt frequency on both systems was scaled by a factor of fifteen to compensate for time dilation. This is a coarse approximation.

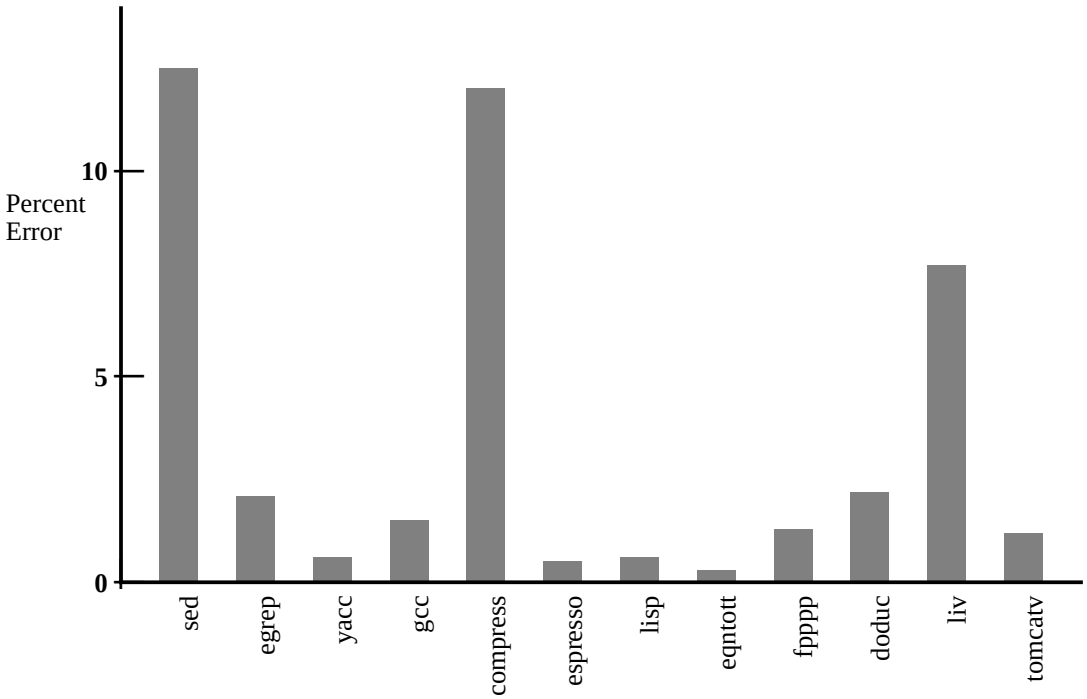

Figure 3: Error in predicted execution times for Ultrix.

The relatively large prediction errors for *sed, compress,* and *liv* are explained by inaccuracies in the simulation model used for prediction. See the text for a complete discussion.

Figure 3 shows percent error for predictions of execution time for twelve workloads running under Ultrix[***]. Predictions for most of the workloads are quite good. Three of the workloads have errors greater than five percent. The explanation for these errors give interesting insight into the behavior of the tracing system:

- *Sed* has the shortest execution time of all the workloads, under 0.5 seconds for three runs. The 12% error corresponds to 0.06 seconds. Such a short execution time exaggerates the distortion introduced by disk latency approximations.
- *Compress* has the largest input file of all the workloads, 100K bytes, but its execution time is only 1.32 seconds. The prediction error is mostly due to disk read-ahead phenomena, where reads-ahead requests to disk complete in the traced system but induce idle time in the untraced system. A comparison of idle time predicted by the

[***]Because of the large variability of running time induced by the Mach 3.0 page mapping policy, we do not present error figures for Mach 3.0.

trace/simulation system and idle time measured by the timing facility in the c-shell [31] confirms that the simulator does under-estimate idle activity.

- *Liv* has the worst write-buffer behavior of all the workloads, and also has significant floating point activity. The prediction error is caused by the overlapping of write buffer and floating point activity that is not modeled in the simulator.

Considering the known sources of error, the estimated execution times correlate well with measurements of execution time made with an accurate timer. Estimates of idle time are one of the dominant sources of error. As idle time has a negligible effect on cache performance, this source of error in execution-time predictions does not cause a significant distortion for simulations of memory system behavior for the restricted class of workloads we consider. Similarly, the simulation does not model the overlap of floating point delays with memory delays, but this has no impact on cache activity, as floating point delay has negligible impact on the pattern of cache misses that occur. Page mapping is another source of error, and the random policy used by Mach 3.0 causes much greater variation in execution times, with a subsequent loss of precision in time predictions. The good estimates of running time for most of the workloads demonstrates that the address trace collection is accurate, with errors in predicted execution time due primarily to inaccuracies in the modeling of the system rather than error inherent in the trace.

5.2. User TLB Miss Count

Using a kernel with a user TLB miss counter, we compared the TLB miss counts predicted by the simulator to TLB miss counts from an uninstrumented system (See Table 3).

	Mach 3.0		Ultrix	
workload	predicted	measured	predicted	measured
sed	7493	6438	131	190
egrep	6430	6122	164	191
yacc	9270	7494	270	318
gcc	53389	48355	29057	29948
compress	91706	89966	79682	79692
espresso	10351	7252	838	1006
lisp	28605	37919	110	179
eqntott	717428	706915	675166	674579
fpppp	22816	21893	3256	1894
doduc	48859	39129	6023	3510
liv	2753	2423	70	63
tomcatv	340968	359976	317872	314950

Table 3: TLB misses, measured and predicted.

One source of error in the TLB miss predictions is explicit TLB writes from the kernel. The kernel sometimes avoids a user TLB miss by writing the TLB explicitly, using `tlbdropin()` in Ultrix or `tlb_map_random()` in Mach. In the simulator, which does not know about these writes, all TLB fills are caused by TLB misses. Kernel instruction reference counts for *gcc* showed about 1800 calls to `tlbdropin()` for Ultrix, and 3700 calls to `tlb_map_random()` for Mach. Also observe that the TLB uses a random replacement policy. The miss rates predicted by the simulator demonstrate a certain amount of error. Given the type of activity and its small impact on overall performance, this error does not detract significantly from the quality of overall measurements. These measurements demonstrate another end-to-end method that was used to evaluate and improve the correlation between simulated and real behavior.

6. Conclusions

Our experience demonstrate that software methods can be applied to collect full address traces, with both system and user references. We have demonstrated instrumentation tools, trace formats and techniques that help insure trace quality, and measurements to establish that address traces reflect true system behavior. Traces from all three systems have already been applied to numerous problems in memory system and software design research [5, 7, 8, 9, 18].

References

[1] Anant Agarwal, Richard L. Sites, and Mark Horowitz.
ATUM: A New Technique for Capturing Address Traces Using Microcode.
In *The Proceedings of the 13th International Symposium on Computer Architecture*, pages 119-127. June, 1986.

[2] Anant Agarwal, John Hennessy, and Mark Horowitz.
Cache Performance of Operating System and Multiprogramming Workloads.
ACM Transactions on Computer Systems 6(4):pp 393-431, November, 1988.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska.
The Interaction of Architecture and Operating System Design.
In *The Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108-120. April, 1991.

[4] Thomas Ball and James R. Larus.
Optimally Profiling and Tracing Programs.
In *Principles of Programming Languages*. January, 1992.

[5] Anita Borg, R.E. Kessler, Georgia Lazana, and David Wall.
Long Address Traces from RISC Machines: Generation and Analysis.
WRL Research Report 89/14, Digital Equipment Corporation Western Research Laboratory, 1989.

[6] David Callahan, Ken Kennedy, and Allan Porterfield.
Software Prefetching.
In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. April, 1991.

[7] J. Bradley Chen and Brian N. Bershad.
The Impact of Operating System Structure on Memory System Performance.
In *Proceedings of the 14th ACM Symposium on Operating System Principles*. December, 1993.

[8] J. Bradley Chen.
Memory Behavior for an X11 Window System.
In *The Proceedings of the Winter 1994 USENIX Conference*. January, 1994.

[9] J. Bradley Chen, Anita Borg, and Norman P. Jouppi.
A Simulation Based Study of TLB Performance.
In *The Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 114-123. May, 1992.

[10] Douglas W. Clark.
Cache Performance in the VAX-11/780.
ACM Transactions on Computer Systems 1(1):pp. 24-37, February, 1983.

[11] Douglas W. Clark, Peter J. Bannon, and James B. Keller.
Measuring VAX 8800 Performance with a Histogram Hardware Monitor.
In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 176-185. June, 1988.

[12] Douglas W. Clark and Joel S. Emer.
Performance of the VAX 11/780 Translation Buffer: Simulation and Measurement.
ACM Transactions on Computer Systems 3(1):270-301, February, 1985.

[13] Robert F. Cmelik and David Keppel.
Shade: A Fast Instruction-Set Simulator for Execution Profiling.
In *The Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*. May, 1994.

[14] J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud.
BACH: BYU Address Collection Hardware; The Collection of Complete Traces.
In *Proceedings of the 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*. 1992.

[15] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith.
Cache Performance of the SPEC Benchmark Suite.
Technical Report, University of Wisconsin-Madison, 1991.

[16] Mark D. Hill.
Aspects of Cache Memory and Instruction Buffer Performance.
PhD thesis, University of California at Berkeley, Computer Sciences Division, November, 1987.
Number UCB/CSD 87/381.

[17] Mark D. Hill.
A Case for Direct-Mapped Caches.
IEEE Computer 21(12):25-40, December, 1988.

[18] R.E. Kessler and Mark D. Hill.
Page Placement Algorithms for Large Real-Indexed Caches.
ACM Transactions on Computer Systems 10(4), November, 1992.

[19] James R. Larus.
personal communication.
June, 1993.

[20] Jeffrey C. Mogul and Anita Borg.
The Effect of Context Switches on Cache Performance.
In *The Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75-84. April, 1991.

[21] 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, November, 1992.
CSE-TR-147-92.

[22] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge and Richard Brown.
Design Tradeoffs for Software-Managed TLBs.
In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 27-38. May, 1993.

[23] David Nagle, Richard Uhlig, Tim Stanley, Trevor Mudge, Stuart Sechrest.
Optimal Allocation of On-chip Memory for Multiple-API Operating Systems.
Technical Report, University of Michigan, 1993.
CSE-TR-184-93.

[24] John K. Ousterhout.
Why Operating Systems Aren't Getting Faster As Fast As Hardware.
In *Proceedings of the Summer 1991 USENIX Conference*, pages 247-256. June, 1991.

[25] Richard F. Rashid, Avadis Tevanian, Jr., Michael Young, David B. Golub, Robert V. Baron, David Black, William Bolosky and Jonathan Chew.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31-39. April, 1987.

[26] Alan Jay Smith.
Cache Memories.
ACM Computer Surveys 14(3):473-530, September, 1982.

[27] Micheal D. Smith.
Tracing with Pixie.
Technical Report, Stanford University, November, 1991.

[28] Smith, J. E. and Goodman, J. R.
Instuction Cache Replacement Policies and Organizations.
IEEE Transactions on Computers 34(3):234-241, March, 1985.
.

[29] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson.
Tradeoffs in Supporting Two Page Sizes.
In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 415-424. June, 1992.

[30] Josep Torellas, Anoop Gupta, and John Hennessy.
Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System.
In *The Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162-174. October, 1992.

[31] *UNIX User's Manual, Supplementary Documents*
USENIX Association, 1984.
An Introduction to the C shell.

[32] Bart C. Vashaw.
Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors.
PhD thesis, Carnegie Mellon University, 1992.
Department of Electrical and Computer Engineering.

[33] David W. Wall.
Global Register Allocation at Link-time.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 264-275. 1986.
Also available as WRL Technical Report 86/3.

[34] David W. Wall and Michael L. Powell.
The Mahler Experience: Using an Intermediate Language as the Machine Description.
In *Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 100-104. 1987.
A more detailed version is available as WRL Technical Report 87/1.

[35] David W. Wall.
Systems for Late Code Modification.
In Robert Giergerich and Susan L. Graham, *Code Generation --- Concepts, Tools, Techniques.*
Springer-Verlag, 1992, pages 275-293.

WRL Research Reports

''Titan System Manual.''
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

''Global Register Allocation at Link Time.''
David W. Wall.
WRL Research Report 86/3, October 1986.

''Optimal Finned Heat Sinks.''
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

''The Mahler Experience: Using an Intermediate Language as the Machine Description.''
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

''The Packet Filter: An Efficient Mechanism for User-level Network Code.''
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

''Fragmentation Considered Harmful.''
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

''Cache Coherence in Distributed Systems.''
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

''Register Windows vs. Register Allocation.''
David W. Wall.
WRL Research Report 87/5, December 1987.

''Editing Graphical Objects Using Procedural Representations.''
Paul J. Asente.
WRL Research Report 87/6, November 1987.

''The USENET Cookbook: an Experiment in Electronic Publication.''
Brian K. Reid.
WRL Research Report 87/7, December 1987.

''MultiTitan: Four Architecture Papers.''
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

''Fast Printed Circuit Board Routing.''
Jeremy Dion.
WRL Research Report 88/1, March 1988.

''Compacting Garbage Collection with Ambiguous Roots.''
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

''The Experimental Literature of The Internet: An Annotated Bibliography.''
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

''Measured Capacity of an Ethernet: Myths and Reality.''
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

''Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.''
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

''SCHEME->C A Portable Scheme-to-C Compiler.''
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

''Optimal Group Distribution in Carry-Skip Adders.''
Silvio Turrini.
WRL Research Report 89/2, February 1989.

''Precise Robotic Paste Dot Dispensing.''
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

''Simple and Flexible Datagram Access Controls for Unix-based Gateways.''
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.''
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.''
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

''A Unified Vector/Scalar Floating-Point Architecture.''
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.''
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.''
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.''
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

''The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.''
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

''Long Address Traces from RISC Machines: Generation and Analysis.''
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

''Link-Time Code Modification.''
David W. Wall.
WRL Research Report 89/17, September 1989.

''Noise Issues in the ECL Circuit Family.''
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

''Efficient Generation of Test Patterns Using Boolean Satisfiablilty.''
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

''Two Papers on Test Pattern Generation.''
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

''Virtual Memory vs. The File System.''
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

''Efficient Use of Workstations for Passive Monitoring of Local Area Networks.''
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

''A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.''
John S. Fitch.
WRL Research Report 90/6, July 1990.

''1990 DECWRL/Livermore Magic Release.''
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

''Pool Boiling Enhancement Techniques for Water at Low Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

''Writing Fast X Servers for Dumb Color Frame Buffers.''
Joel McCormack.
WRL Research Report 91/1, February 1991.

''A Simulation Based Study of TLB Performance.''
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

''Analysis of Power Supply Networks in VLSI Circuits.''
Don Stark.
WRL Research Report 91/3, April 1991.

''TurboChannel T1 Adapter.''
David Boggs.
WRL Research Report 91/4, April 1991.

''Procedure Merging with Instruction Caches.''
Scott McFarling.
WRL Research Report 91/5, March 1991.

''Don't Fidget with Widgets, Draw!.''
Joel Bartlett.
WRL Research Report 91/6, May 1991.

''Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

''Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.''
G. May Yip.
WRL Research Report 91/8, June 1991.

''Interleaved Fin Thermal Connectors for Multichip Modules.''
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

''Experience with a Software-defined Machine Architecture.''
David W. Wall.
WRL Research Report 91/10, August 1991.

''Network Locality at the Scale of Processes.''
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

''Cache Write Policies and Performance.''
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

''Packaging a 150 W Bipolar ECL Microprocessor.''
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

''Observing TCP Dynamics in Real Networks.''
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

''Systems for Late Code Modification.''
David W. Wall.
WRL Research Report 92/3, May 1992.

''Piecewise Linear Models for Switch-Level Simulation.''
Russell Kao.
WRL Research Report 92/5, September 1992.

''A Practical System for Intermodule Code Optimization at Link-Time.''
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

''A Smart Frame Buffer.''
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

''Recovery in Spritely NFS.''
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

''Tradeoffs in Two-Level On-Chip Caching.''
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

''Unreachable Procedures in Object-oriented Programing.''
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

''An Enhanced Access and Cycle Time Model for On-Chip Caches.''
Steven J.E. Wilton and Norman P. Jouppi.
WRL Research Report 93/5, July 1994.

''Limits of Instruction-Level Parallelism.''
David W. Wall.
WRL Research Report 93/6, November 1993.

''Fluoroelastomer Pressure Pad Design for Microelectronic Applications.''
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

''A 300MHz 115W 32b Bipolar ECL Microprocessor.''
Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.
WRL Research Report 93/8, December 1993.

''Link-Time Optimization of Address Calculation on a 64-bit Architecture.''
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

''ATOM: A System for Building Customized Program Analysis Tools.''
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

''Complexity/Performance Tradeoffs with Non-Blocking Loads.''
Keith I. Farkas, Norman P. Jouppi.
WRL Research Report 94/3, March 1994.

''A Better Update Policy.''
Jeffrey C. Mogul.
WRL Research Report 94/4, April 1994.

''Boolean Matching for Full-Custom ECL Gates.''
Robert N. Mayo, Herve Touati.
WRL Research Report 94/5, April 1994.

''Software Methods for System Address Tracing: Implementation and Validation.''
J. Bradley Chen, David W. Wall, and Anita Borg.
WRL Research Report 94/6, September 1994.

''Performance Implications of Multiple Pointer Sizes.''
Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.
WRL Research Report 94/7, December 1994.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter’’
David Boggs.
WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package’’
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics’’
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0’’
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture’’
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors’’
Scott McFarling.
WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates’’
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.

‘‘Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS Circuits’’
Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.
WRL Technical Note TN-45, March 1994.

Table of Contents

List of Figures

List of Tables