The Impact of Software Structure and Policy on CPU and Memory System Performance

J. Bradley Chen

May 1994

CMU-CS-94-145

Carnegie Mellon

0

The Impact of Software Structure and Policy on CPU and Memory System Performance

J. Bradley Chen
May 1994
CMU-CS-94-145

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890

Submitted in partial fu[illegible]llment of the requirements for the degree of Doctor of Philosopy.

Thesis Committee:
J. Douglas Tygar, Chair
Brian N. Bershad
Daniel P. Siewiorek
Thomas R. Gross
Anita Borg, Digital Equipment Corporation

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, the Advanced Research Projects Agency under contract number F19628-93-C-0193, the National Science Foundation under Presidential Young Investigator Grant CCR-8858087, Digital Equipment Corporation, Motorola, and TRW.

Keywords: operating systems, memory systems, performance, address tracing

arnegie
ellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

The Impact of Software Structure and Policy on CPU and Memory System Performance

J. BRADLEY CHEN

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Accesion For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

ACCEPTED:

[illegible]
THESIS COMMITTEE CHAIR **DATE**

[illegible] 5/26/94
DEPARTMENT HEAD **DATE**

APPROVED:

[illegible] 5/27/94
DEAN **DATE**

Abstract

Operating systems, when compared to application programs, have received disappointingly little benefit from the performance improvements of the most recent generation of microprocessors. This thesis used complete traces of software activity from a RISC-based uniprocessor to expose the dynamic behavior of operating system execution and explore the sources of poor performance. Traces from both Mach 3.0 and Ultrix implementations of UNIX permitted a study of performance differences between microkernel and monolithic implementations of the same operating system interface. The comparison showed that both system structure and policy implemented in the system have a significant impact on performance.

Measurements of X11 workloads showed that memory system behavior for these large workloads differs significantly from the kinds of workloads traditionally used for performance analysis. Structural and behavioral similarities between large X11 workloads and the operating system are reflected in their overall performance.

These experiments represent the first application of address tracing with software instrumentation to the study of behavior in a popular operating system. The development of the tools and their application in this research demonstrates that such methods are a powerful tool for understanding behavior and interactions in complex software systems. Overall, the experiments demonstrate that system level effects must be measured for a complete understanding of overall performance.

Acknowledgements

In the few words that this acknowledgement permits, there is no way to adequately recognize the contributions of all the people who have helped me in my thesis work. Here I briefly mention a few of the names. The true evidence of their contributions should be sought not here but in the document that follows.

Doug Tygar, as my advisor, has shown great wisdom, judgement, and patience in guiding me through the Ph.D. program at Carnegie Mellon.

Anita Borg has been my mentor since my first internship in at the DEC Western Research Laboratory in the summer of 1986. Without her involvement this project would never have happened.

Brian Bershad taught me invaluable lessons on the meaning of quality research.

Thanks to David Wall for providing the epoxie tool, and for his continued support throughout this project. I'd also like to thank the entire staff of the DEC Western Research Laboratory, past and present, for showing me how fun and exciting computer systems research can be.

Thanks to the School of Computer Science of Carnegie Mellon University, under the leadership of Raj Reddy, Jim Morris, and the late Nico Haberman. I am honored to have had the opportunity to be a part of the long tradition of excellence in systems research for which this department is renowned. I look forward to seeing this tradition continue.

I'd like to acknowledge some of the other members of the CMU SCS community for putting up with me and helping me along the way. From the Mach group: Bob Baron, Sandro Forin, Dan Stodolsky, Mary Thompson, and Terri Watson. From the Coda group: Satya, David Steere, Brian Noble, and Josh Raiff. From SCS Facilities: Mark Puskar, Dave Livingston, and Gary Wallace. Also thanks to Anna Marie Mackuliak, Marian D'Amico, Laura Forsyth, Maggie Muller, Angie Brookins, Jim Skees, Sharon Burks, and Catherine Copetas.

This thesis is dedicated to my parents.

Table of Contents

1. Introduction 1

2. Background 7

2.1. Related Work 7

2.2. Mach 3.0 and Ultrix 17

2.2.1. Similarities between Mach 3.0 and Ultrix 19

2.3. Workloads 20

2.4. Conclusions 20

3. The Tracing System 23

3.1. The Address Tracing System 23

3.1.1. Epoxie 25

3.2. Tracing the Kernel 28

3.2.1. Implementation details for the two systems 29

3.3. The DECstation 5000/200 Memory System 31

3.4. Ensuring Trace Quality 32

3.4.1. Avoiding Trace Distortion 33

3.4.2. Defensive Tracing 35

3.5. Validation of Methods 36

3.5.1. Program Execution Time 36

3.5.2. User TLB Miss Count 40

3.5.3. Explaining Anomalous Behavior 41

3.6. Experience with Mach 3.0 and Ultrix 43

4. Mach 3.0 and Ultrix: The Impact of Structure 47

4.1. Isolating the Effects of Structure 47

4.2. Comparative System Behavior 48

4.2.1. Breakdown of System Activity 50

4.3. Memory System Behavior: Seven Assertions 56

4.3.1. System and user locality 57

4.3.2. System instruction locality 58

4.3.3. Competition between user and system activity 59

4.3.4. System self-interference 62

4.3.5. Block operations 64

4.3.6. Streaming writes 66

4.3.7. Page mapping policy 68

4.4. Conclusions 69

5. Mach 3.0 and Ultrix: The Impact of Policy 73

5.1. The Impact of Structure on Execution Time 73
5.2. Disk Policy 75
5.3. Page Mapping Policy 76
5.4. Conclusions 80
6. X11 Workloads 81
6.1. Background 81
6.1.1. Related Work in Measuring X11 Performance 83
6.2. Workloads 84
6.3. Experiments and Analysis 85
6.3.1. Memory Cycles Per Instruction 85
6.3.2. Cache Effects 87
6.3.2.1. Inter-Context Competition 87
6.3.2.2. Self-Interference misses 89
6.3.2.3. Summary 90
6.3.3. TLB Behavior 91
6.4. Conclusions 92
7. TLB Behavior 95
7.1. Introduction 95
7.1.1. Workloads and Methodology 98
7.2. Instruction TLB results 99
7.2.1. Micro-TLBs 100
7.2.1.1. The Impact of System Activity 107
7.2.2. Instruction TLBs 108
7.2.2.1. The Impact of System Activity 112
7.3. Data and Shared TLB Results 113
7.3.1. The Impact of System Activity 118
7.4. Variable Size TLB Entries 121
7.5. Conclusions 125
8. Conclusions 127
8.1. Summary 127
8.1.1. Making generalizations from this work 129
8.2. Evaluation 131
8.2.1. Mach 3.0 and Ultrix 131
8.2.2. The DECstation 5000/200 Memory System 132
8.3. Implications for Current and Future Work 133
8.3.1. Software 133
8.3.2. Hardware 134
8.4. Conclusion 135
References 137
Appendix A. Hardware Specifications 145
Appendix B. Software Specifications 147
Appendix C. List of Abbreviations 155

List of Figures

Figure 3-1:	Overview of the tracing system.	24
Figure 3-2:	Instrumentation by epoxie	26
Figure 3-3:	Error in predicted execution times for Ultrix.	39
Figure 4-1:	Baseline MCPI for Ultrix and Mach.	51
Figure 4-2:	Relative system overheads for programs running on Ultrix and Mach.	53
Figure 4-3:	System instruction cache misses for Ultrix and Mach.	55
Figure 4-4:	System data cache misses for Ultrix and Mach.	56
Figure 4-5:	User/system interference.	60
Figure 4-6:	System self-interference.	63
Figure 4-7:	MCPI for random page mapping.	67
Figure 5-1:	Ultrix and Mach 3.0 Execution Time: The Impact of Structure	74
Figure 5-2:	Page Mapping Policy and Mach 3.0 Execution times.	77
Figure 5-3:	Ultrix and Mach 3.0 Execution Time; Page mapping policy as implemented.	79
Figure 6-1:	Baseline MCPI for X11 Workloads.	86
Figure 6-2:	*MCPI* with and without inter-context competition.	88
Figure 6-3:	Inter-Context Competition, Associativity, and *MCPI*.	90
Figure 7-1:	*Eqntott* micro-TLB behavior	100
Figure 7-2:	Micro-TLB Behavior, 4K byte page	102
Figure 7-3:	SPECmark micro-TLB Behavior, 16K byte pages	104
Figure 7-4:	LRU vs. FIFO Replacement	105
Figure 7-5:	Estimating micro-TLB CPI Contribution for *gcc*	106
Figure 7-6:	*Gcc* Instruction TLB Behavior	110
Figure 7-7:	Multi-task instruction TLB behavior	111
Figure 7-8:	*Mat300* Shared TLB Behavior	114
Figure 7-9:	*Tree* Shared TLB Behavior	115
Figure 7-10:	Multiprocess Mix TLB Behavior	116
Figure 7-11:	Shared vs. Split TLBs	119
Figure 7-12:	*mat300* Data Memory Access Patterns	122
Figure 7-13:	*tree* Data Memory Access Patterns	123
Figure 7-14:	*gcc* Instruction Memory Access Patterns	124

List of Tables

Table 1-1: Seven assertions about the memory behavior of operating systems. 3
Table 2-1: Experimental workloads with execution times for a DECstation 5000/200. 21
Table 3-1: Memory system simulation parameters. 32
Table 3-2: Run Times, measured and predicted, in seconds. 37
Table 3-3: TLB misses, measured and predicted. 41
Table 4-1: Summary of trace results. 49
Table 4-2: Summary penalty cycles (per instruction). 49
Table 4-3: MCPI contributions from the cache. 59
Table 4-4: MCPI contributions from cache competition. 61
Table 4-5: TLB activity. 62
Table 4-6: MCPI from block memory operations. 65
Table 4-7: Block memory operations and memory reads. 65
Table 4-8: Write buffer stall cycles per instruction. 66
Table 6-1: Experimental workloads. 84
Table 6-2: Instruction and Data Reference Counts 85
Table 6-3: TLB Misses per 1000 instructions. 91
Table 7-1: SPECmark FIFO micro-TLB Miss Rates 106
Table 7-2: Micro-TLB Miss Rates for Mach 3.0 108
Table 7-3: Micro-TLB Miss Rates for DEC Ultrix 108
Table 7-4: Instruction TLB Miss Rates 112
Table 7-5: Instruction TLB Miss Rates for Mach 3.0 112
Table 7-6: Instruction TLB Miss Rates for DEC Ultrix 113
Table 7-7: Shared TLB Miss Rates. 117
Table 7-8: Data TLB Miss Rates. 118
Table 7-9: Data Miss Rates for Mach 3.0 119
Table 7-10: Data Miss Rates for DEC Ultrix 120
Table 7-11: Shared Miss Rates for Mach 3.0 120
Table 7-12: Shared TLB Miss Rates for DEC Ultrix 120

Chapter 1

Introduction

Operating systems, when compared to application programs, have received disappointingly little benefit from the performance improvements of the most recent generation of microprocessors [6, 63]. This reflects a lack of understanding among operating system designers and computer architects as to the behavior of operating system execution on machines with hierarchical memory systems. System execution differs in important ways from user execution, leading to worse-than-expected performance for system-intensive workloads on current machines. As CPU performance improves and the ratio of processor speed to memory access time increases, system performance becomes increasingly sensitive to system organization and policy.

The goal of this thesis is to address some present shortcomings in our understanding of system behavior. This thesis will show that, for a RISC-based UNIX workstation, the structure of software systems and policy implemented in the systems have a large impact on performance at the level of the hardware-software interface, both in terms of memory system behavior and raw instruction counts. It also shows that these low-level system effects must be measured for a complete understanding of overall performance.

The four main contributions of this thesis are:

- An evaluation of the impact of operating system structure on overall performance, founded on the comparison of two popular contemporary operating systems: one with monolithic structure and one structured as a microkernel.
- An evaluation of the impact of some system policy issues on overall performance.
- An evaluation of memory system behavior for an important class of workloads that is often neglected in performance evaluations.
- A demonstration of new tools and methodologies for making complete and accurate measurements of hardware/software interaction, using traces of memory reference behavior.

In this research we consider microkernel operating systems, where significant operating system functionality is moved out of the kernel address space and into a user-level server process. We also consider program execution on a RISC-based uniprocessor. Measurements for RISC uniprocessors and microkernel systems are lacking in prior research. Overall, system activity has been neglected in many recent studies because of the inadequacy of prior techniques for making measurements on new machines, and because of the complexity involved in applying more recent methods. The measurement systems developed for this thesis use novel software-based techniques that scale with changes in hardware technology.

Because of the lack of quantitative measurements, operating system designers have often relied on beliefs and intuitions when evaluating the performance impact of system design choices. One of the original goals of this research was to investigate some of these beliefs in the context of the two operating systems. We have collected some of the beliefs in Table 1-1, and will use them as a basis of our comparison of memory system performance in two implementations of the UNIX operating system: Ultrix from Digital Equipment Corporation [79], and Mach 3.0 from Carnegie Mellon University [1, 10]. The beliefs in Table 1-1 are stated as assertions along with their impact for system designers. These assertions are derived from the computer systems literature and are based on past experiences [31], measurements of microbenchmarks [15, 63], and extensive measurements of real systems [3, 4, 6, 24, 26, 57, 78, 82]. This thesis will explore behavior related to the beliefs in Table 1-1, for truth as well as performance impact. This exploration will demonstrate that system designers have an inadequate and sometimes incorrect understanding of the performance impact of system structure for a current uniprocessor-based computer.

The cross-system comparison built around the assertions in Table 1-1 focuses primarily on system structure and its impact on execution time. However, overall performance depends not only on system structure but also on policy implemented in the operating system. To complete a comprehensive investigation of overall performance, this thesis includes a discussion of two system policy issues — disk management and virtual-to-physical page map policy — which are different for Ultrix and Mach 3.0 and which significantly impact performance.

Assertion	Implication
1. The operating system has less instruction and data locality than user programs [24, 26].	The operating system isn't getting faster as fast as user programs.
2. System execution is more dependent on instruction cache behavior than is user execution [78].	A balanced cache system for user programs may not be balanced for the system.
3. Collisions between user and system references lead to significant performance degradation in the memory system (cache and TLB) [60, 78, 82].	A split user/system cache could improve performance.
4. Self-interference is a problem in system instruction reference streams [57, 78].	Increased cache associativity and/or the use of text placement tools could improve performance.
5. System block memory operations are responsible for a large percentage of memory system reference costs [63, 78].	Programs that incur many block memory operations will run more slowly than expected.
6. Write buffers are less effective for system (as opposed to user) reference streams [6, 34].	A write buffer adequate for user code may not be adequate for system code.
7. Virtual page mapping strategies can have significant impact on cache performance [47, 59].	Systems should support a flexible page mapping interface, and should avoid default strategies that are prone to pathological behavior.

Table 1-1: Seven assertions about the memory behavior of operating systems.

The workloads used in the comparison of Mach 3.0 and Ultrix were selected from the benchmarks commonly used to evaluate computer system performance. Although these workloads have been selected and developed to identify strengths and weaknesses of computer systems, they are not necessarily representative of the kind of applications for which these machines are typically used [58]. To investigate the behavior of some more popular applications, we measured the activity of workloads that use the X11 windowing system. This thesis shows that X11 workloads differ in important ways from traditional benchmarks, that these differences impact overall performance, and that these differences have implications for memory system design.

The measurements presented in this thesis were made using trace-based simulation. The tracing system was developed for this thesis and made it possible to collect detailed measurements of software behavior as a complete trace of memory references for both user and system activity. These traces drove simulations of performance-critical hardware structures such as caches whose behavior can't be measured directly.

The development of the tracing system was a significant part of this thesis research. The tracing system uses software methods to collect complete address trace data on a DECstation 5000/200 workstation. It is novel in that it is the first time such methods have been applied to a commercial RISC-based uniprocessor computer or a widely used operating system. Another novel aspect of the tracing system is the methods developed and applied to establish the quality of the trace data it generates. The traces facilitated detailed measurements of system activity not previously available for RISC uniprocessors or for microkernel-based systems. In particular, they provided data about system instruction counts, memory system penalties, and system policy effects that are necessary for a complete understanding of overall performance.

Both the comparative study of Mach 3.0 and Ultrix and the exploration of behavior of X11 workloads used a simulator for the DECstation 5000/200 memory system. Simulation of a complete memory system made it possible to measure all system-induced memory delays, rather than limiting measurements to delays from some isolated memory system component. The tracing methods were also used in a study of translation lookaside buffer (TLB) behavior, which is included in this thesis. Unlike the rest of the thesis, which concentrates on software structure, the TLB study focuses on hardware structure, and demonstrates how the tracing systems can be applied to problems in hardware design.

An important contribution of this research has been in demonstrating the strength of the methods for revealing and explaining counter-intuitive system behavior. In this document we discuss a number of examples. In Chapter 7 we show that higher system instruction counts in Mach 3.0 as compared to Ultrix can lead to better data TLB miss rates Mach 3.0, contradicting a popular intuition than microkernel systems have worse TLB behavior. In Chapter 4 we show that the microkernel structure of Mach 3.0 does not induce significant competition between user and system contexts, contradicting another long-standing assertion. Although user/system interaction across a UNIX interface has a small impact on performance, the impact for client/server interaction across an X11 interface can be much larger, as will be seen in Chapter 6. In Section 4.3.7 we show that the deterministic page mapping policy used by Ultrix will frequently induce worse cache performance than a random policy.

This research differs significantly from prior work in the following:

- *RISC processor architecture:* Earlier trace-based studies of operating systems are founded on microcoded and multi-cycle per instruction architectures [2, 3, 4, 24, 26, 82]. RISC computer architecture is substantially different from earlier design, and mandates an update of prior work.

- *Uniprocessor systems and workloads:* This study focuses entirely on uniprocessor systems and workloads. Despite the performance potential of multiprocessor systems, uniprocessors continue to dominate computing, a situation which appears likely to continue. Several earlier studies [78, 82] emphasize multiprocessor specific systems and workloads.

- *Multiple Operating Systems:* Working with Ultrix and Mach 3.0 made it possible to extend analysis beyond what is possible in a study of a single system. It enabled a comparison of the dynamic behavior of microkernel and monolithic systems. More generally, it made it possible to identify and compare issues that differentiate the two systems and compare their effect on performance. Finally, it allowed us to identify performance problems common to both systems, an indication of how current memory system designs are ill-suited for operating system execution.

  Previous studies that consider operating system activity concentrate on variations in memory system structure [2, 3, 4, 16, 23, 57, 66]. [illegible]ough several [2, 3, 4, 66] used address traces of both DEC Ultrix and [illegible] VMS operating systems, the data from the two systems was used primarily as a basis for identifying common behavior in both operating systems without drawing distinctions between them.

- *Simulation of complete memory systems:* Previous studies have concentrated on memory system components such as caches or TLBs in isolation, varying parameters such as cache size in an attempt to optimize behavior over a given workload set [3, 23, 38, 60]. Simulating a complete memory system has two important advantages. First, it permits an objective evaluation and comparison of operating system memory performance, considering the delays from all memory system components and not just one. Second, it permits a comparison of the relative importance of various memory system components in overall performance.

- *Diverse workloads:* Many prior studies hide differences in behavior between workloads by averaging measurements or only reporting partial results. This can be misleading as activity in the operating system varies significantly with different user workloads. In this study we have tried to expose and explain measured differences between workloads.

In the next chapter we discuss related work, and describe in detail the systems to be measured. Chapter Three gives more detail on the address tracing system, including a discussion of the methods used to measure and improve the quality of the address trace data.

The introductory material is followed by four chapters of analysis. Chapter Four contains a broad comparison of DEC Ultrix and Mach 3.0, exploring how the differences in structure between the two systems have impact on performance, both in terms of instruction counts and memory system latencies. Chapter Five completes the discussion of Mach 3.0 vs. Ultrix performance, using execution time as a metric of overall performance, and focusing on the impact of system policy. In Chapter Six we look at X11 applications running under Ultrix, to explore how the differences in activity and structure affect the behavior of these workloads relative to the workloads that are more traditionally studied. Chapter Seven explores issues in TLB design, both in terms of user-level activity and performance variations induced by the structural differences between Ultrix and Mach 3.0. It is an example application of the tracing methodology to a hardware design problem. Chapter Eight closes with a final summary and conclusions.

Chapter 2
Background

This chapter places the work for this thesis in the context of current and prior research. The first section discusses related work in tracing and measurement systems. This is followed by background information on the software systems measured for this study — first comparing Mach 3.0 and Ultrix, then giving details on the workloads used in the cross-system comparison.

2.1. Related Work

In this section we discuss some of the prior work that has influenced the experiments and measurement systems developed for this thesis. We start by giving an overview of earlier work in understanding operating system behavior. This is followed by a discussion of tradeoffs between hardware and software measurement techniques. We consider current and previous measurement systems to explore some of the issues that influenced the choice of software methods for these experiments. Next we look at some recent hardware and software tracing projects and systems that explore memory system behavior with measurements other than address traces.

Comparing System and User Behavior

System execution and user execution differ in important ways. Some of the differences, such as exception handling and the absence of floating point activity, are easy to recognize. Others, such as the differences in locality patterns between operating system activity and user workloads, are more subtle. Active code in the operating system is spread throughout sparse program text, inducing locality patterns that differentiate reference patterns in the kernel from typical user behavior. As an example, consider an operating system with one megabyte of kernel text running on a uniprocessor computer with a 64K byte direct-mapped virtual instruction cache. In this situation, each kernel instruction overlaps with sixteen other kernel instructions in the cache. In contrast, many of the SPEC workloads [76] have less than 64K bytes of text. User text for such a workload will fit into the direct-mapped virtual cache with no collisions. The probability of user instruction collisions is zero, but the probability of collisions between system instructions is much higher. Such observations are consistent with our measurements (and those of others [3, 24, 78]) which indicate that cache behavior for system code is substantially worse than for user code. System data is also sparse as compared to user data. Control and data reference patterns for the operating system are generally more complex than those of user execution, and this is reflected in cache behavior.

Time spent in the operating system serving application requests is often an important component of overall performance. This observation is substantiated by the measurements made for this thesis and also by prior work. In the DEC Ultrix workloads measured by Agarwal et al., system references account for up to 50% of total memory activity [3]. Recent studies [6, 57, 63] document poor system performance, asserting that the penalties from system effects have been underestimated. Despite the well known differences between system and user execution, and the significant amount of overall time spent in the system, many studies of memory behavior disregard system effects [9, 20, 38, 48, 57, 74, 75]. These studies can be misleading because for many workloads system activity is the principal source of memory delays for modern machines.

Differences between system and user memory behavior are documented in prior research. Clark and Emer [26] identified three differences between system and user execution:

- System code and data is bigger than user code and data.
- System data structures are more complex and pointer rich.
- System loops have fewer iterations.

A later study [3] offered evidence that the first and third items apply to caches but that the second does not.

The coding style and functionality of operating systems has not changed significantly since these earlier studies, so the poor cache and TLB behavior they document can be expected to continue. However, the size and complexity of non-system code has increased significantly. An example is server-based systems, such as the X11 workloads discussed in Chapter 6, which often have complexity and text size comparable to an operating system, and event-driven activation patterns, as in the operating system. Other examples are PC-based interactive applications, such as spreadsheets and document preparation systems, and object-based systems. Such workloads can have code and data of size and complexity comparable to the operating system. As bigger and more demanding applications come into common use, the behavioral differences between system execution and user execution become less distinct. This suggests that the memory reference behavior of large user-level systems should become more like that of the operating system, a conclusion that finds support among the results of this thesis. Unfortunately, computer performance evaluation is currently dominated by such workloads as the SPECmarks, which, in structure and code complexity, are often difficult to distinguish from the workloads common on an earlier generation of machines.

Tradeoffs in Hardware and Software tracing

In their study of VAX 11/780 TLB behavior [26], Clark and Emer used both software simulation and direct hardware measurement. Their contributions include several insightful observations about the tradeoffs between hardware measurements and software simulation. These tradeoffs are repeated here, stated in terms of advantages and disadvantages of software-based methods:

1. *Advantage: No special hardware is required.* Hardware measurement requires "some sort of instrument" that makes an electronic connection to the physical machine.
2. *Advantage: Flexibility.* Data from software methods can be used to drive an arbitrary simulator, permitting more flexibility in the systems that can be studied. Direct measurement is limited by the hardware of the measured machine.
3. *Advantage: Repeatability.* It is possible with software methods to repeat a simulation with the same data set, or use a given data set as input for multiple experiments. Repeatability in direct measurement is precluded by non-determinism in hardware and software.
4. *Disadvantage: Can be inaccurate.* Simulators can become complicated and difficult to debug, particularly as they attempt to closely model the behavior of real hardware.
5. *Disadvantage: Big data.* Modeling of certain hardware components may require an amount of data that makes software methods impractical.
6. *Disadvantage: Workload.* It may be difficult to create simulator input representative of a realistic or even interesting workload.

7. *Disadvantage: Computational resources.* Simulated hardware typically runs an order of magnitude slower than real hardware.

Computer architecture has evolved significantly since these observations were originally presented. The essence of the observations remains valid: that hardware and software methods have strengths and weaknesses, and that they tend to complement each other. However, as methods evolve and computational and storage capacities of machines change, some observations need revision.

A number of properties of modern computer architectures create new, substantial obstacles for hardware methods. This tends to complicate methodologies that rely on hardware support and increase the importance of the advantage for software methods in the first observation. One such property is shrinking cycle time. A hardware monitor is ultimately limited by the speed at which events can be measured and recorded. For the machine that Clark and Emer studied, memory was faster than the processor, and it was straightforward to build a device that could write several words of trace for every machine cycle. CPU speed has grown much more quickly than memory speed since the time of their study, so building such a measurement instrument is no longer straightforward. Hardware monitoring systems do not scale with CPU speed. As machine cycle times decreases they become increasingly difficult to implement. This disadvantage of hardware methods translates to an advantage for software methods, as they tend to scale with the execution speed of the subject machine. They benefit fully from the performance of the memory hierarchy of the traced machine, while avoiding the engineering effort required to build such a memory system.

The development of special-purpose hardware measuring devices is also discouraged by the shrinking time between generations of new machines. This tends to decrease the useful lifetime of such a device, and consequently time available for its development.

Another aspect of modern architectures that poses an obstacle to hardware measurement is the movement of more functionality inside of sealed chip packages. This limits the signals that are realistically available for measurement to those appearing as output pins on the chip package. A straightforward example is on-chip caches. Normally an on-chip cache will prevent the majority of memory references from ever appearing on chip output pins. On-chip caches must be disabled if a complete address trace is to be collected by a hardware monitor. This results in a slowdown of ten or more cycles per memory reference [5, 56], which is of similar magnitude to the time penalty for software methods.

Architectural features such as pipelining complicate the situation further. To illustrate, consider the problem of determining when an instruction address appearing on the chip output pins corresponds to an instruction fetch. Often the signal on the pins will not correspond to an instruction fetch. This occurs during a memory system stall or when the pipeline is being cleared to handle an exception. Thus, it is necessary to trace and interpret additional output signals to know when an instruction fetch has occurred. Even after an instruction fetch has been identified, it can be difficult to determine if the instruction was executed, or if it was aborted for some reason before reaching the end of the pipe. Note that hardware monitors detect hardware events such as bus cycles, while software methods record executed instructions. Hardware event monitors are most useful for precisely characterizing a specific machine. Software methods provide traces at a level of abstraction and generality appropriate for modeling hypothetical memory systems.

Due to the need of large traces [16], most recent software tracing systems, including the system developed for this dissertation, use the trace as it is generated, rather than storing it to stable media. In this way, the problem of big data (Observation 5 above) has been partially overcome. It also follows that the desirable property of repeatability (Observation 3) does not apply to these recent software tracing systems.

Repeatability is useful for debugging and can be useful when comparing different simulated hardware, but it does not reflect an accurate system model. Real computer systems have non-determinism, and hence are not perfectly repeatable. For the experiments discussed in this dissertation, the variation between runs was minimized by making experiments sufficiently long, such that overall trends in behavior dominate the variations due to non-determinism.

The impact of Observation 7, relating to the computation requirements of hardware simulation, is subject to three trends that are somewhat interrelated: the increasing speed of computer systems, the increasing size of structures (caches etc.) to be simulated, and increasing resource requirements of workloads of interest. Two remarks are in order. First, with address trace data, it takes much longer to analyze a word of data than it does to generate it, so the speed of hardware data-generation is of little benefit when software methods must be used to analyze the data. Second, with current technology and current workloads a simulation-based study is feasible, as demonstrated by the present work. In the systems used for the present research, the scaling factor for real time to simulated time (trace generation+analysis) is about 100.

New Tradeoffs for Hardware/Software Systems

The seven observations by Clark and Emer are based on measurement systems available in 1985. More recent developments in software methods suggest several new observations on the tradeoffs between hardware and software methods for address tracing.

Compactness of trace is a useful property of many recent software address tracing systems that is difficult to achieve with hardware tracing. Most software tracing systems generate only one instruction address per basic block executed, significantly reducing the amount of instruction trace generated. An example is the QPT system from the University of Wisconsin, which analyzes the control-flow graph of a program for optimal placement of instrumentation, reducing slow-down for traced code and the size of trace that is generated. [9] Although compact trace might be possible for a hardware system, no such system is known, and the additional expense involved in building such hardware may make it impractical.

Another advantage of software methods is the ease of integrating useful supplemental information into the trace as it is recorded. An example is virtual-to-physical page mapping information. Ideally, both physical and virtual addresses should be available in address trace data. Physical addresses are needed to model the behavior of data in physically addressed memory structures. Virtual addresses and address space identifiers are needed to determine for a given address the precise instruction or data item being referenced. Virtual addresses are sufficient when the algorithm that creates the virtual to physical mapping of interest can be simulated. Although this information can be obtained in hardware tracing systems, it generally requires either software support or additional hardware complexity. The information is available in a relatively straightforward way when software methods are used.

Software methods provide both physical and virtual addresses. In the system developed for this thesis, a trace of virtual addresses is augmented with page map and context switch information so that physical page mappings and physical addresses can also be determined.

With a hardware tracing system, obtaining both physical and virtual addresses will generally be more difficult. Depending on hardware and monitoring equipment, an address trace may include virtual or physical addresses, but will generally not include both. This means that page mappings must be extracted by some other method. There are two possibilities: use software support or trace another bus. Either method will complicate hardware trace collection.

A final advantage of software tracing systems is the relatively low cost of duplicating the system. Software tracing does not require custom hardware or special-purpose measurement devices, so the cost of duplicating a software tracing system is potentially much lower than the cost for duplicating a hardware system. An example is the tracing system developed for this thesis, which is currently being used for several different projects both within and outside of Carnegie Mellon University.

Hardware Address Tracing

A review of recent hardware tracing projects demonstrates how the obstacles of hardware tracing have limited its applicability in current research.

In ATUM, the microcode of a DEC VAX 8200 was modified to record an address trace of system and user execution[1] [3]. ATUM was demonstrated to be a valid technique for producing trace on microcoded machines. The system was used to study both VMS and Ultrix systems and a broad range of cache configurations.

ATUM has several limitations. The foremost is that it can only be used on microcoded processors. Recent trends towards RISC processor design suggest that microcode has little application in future processor designs. Another limitation of the system is that long contiguous trace was not possible. The ATUM designers proposed *trace stitching* to address this limitation.

Researchers at Stanford University used a hardware monitor to trace cache misses on a Silicon Graphics multiprocessor based on the MIPS R3000 microprocessor [78]. With

[1]In as much as microcode can be considered a part of computer hardware, and the software running on the traced system was unmodified, we classify this as a hardware method.

their system they determined that a relatively small amount of system code and data were responsible for a large proportion of system cache misses. They also investigated issues related to multiprocessors. An important advantage of their technique is that tracing events are infrequent as compared to full reference traces, making the application of hardware monitors less difficult. A limitation of the technique is that only misses, not references, are recorded. This hides much interesting information, such as cache miss rates, TLB/write buffer behavior, and dynamic instruction frequency. It also makes it impossible to study behavior of caches smaller than those of the traced machine.

The Monster system from the University of Michigan uses a logic analyzer to capture signals from the CPU chip of a DECstation 3100 [59]. They have used their system for extensive studies of system TLB behavior [60]. A limitation for this system is the amount of contiguous trace that can be collected in the logic analyzer, and also the difficulty in interpreting signals on chip output pins in real time.

The BACH system, developed at Brigham-Young University, has been used to collect address trace on both Intel 80486 and Motorola 68030 based machines [36]. They use very fast hardware to record signals from processor output pins at full clock speed. A high-priority interrupt is used to suspend the traced system while address trace is being processed. Their system is designed to work for processors with a clock rate of up to 25 MHz. They have used measurements from their system to argue that system behavior must be measured to achieve accurate results for some workloads. They have made a number of short traces of an Intel 80486 system publicly available. They are reported to be working on a tracing system for a SPARC based Sun workstation. To date they have not demonstrated their system for RISC-based machines or machines with faster clock rates.

Software Address Tracing

For our research, a tool called *epoxie* from DEC Western Research Laboratory was used to instrument executables so that address trace is generated as a side effect of program execution. Epoxie borrows from a previous DEC WRL tracing system that ran on the DEC WRL Titan computer [61]. It was based on load-time modifications in an intermediate language called Mahler [16, 84]. The WRL systems and the system used in this research both use the operating system to control the tracing experiment and manage trace data. This makes it possible to handle trace activity from multiple concurrent contexts, and to use arbitrarily long trace experiments by analyzing address trace as it is generated.

Epoxie instrumentation is similar to that of the *pixie* tool from MIPS Computer Systems [75]. Both use *register stealing* to allocate registers for the tracing system after compilation and program loading. There are several differences between the two tools. When code is instrumented for tracing, jump and branch offsets must be adjusted for the new binary. With epoxie, relocation information is required, and the loader is used after instrumentation to adjust the offsets. Pixie avoids the relocation step by using a jump table the size of the original executable. The jump table is used to look up corrections to jump offsets made necessary by instrumentation. In using a jump table, Pixie doesn't need relocation information.

Another difference between epoxie and pixie is that pixie uses a file descriptor for trace output, while code instrumented with epoxie writes trace into special pages reserved in user memory. The epoxie mechanism is better suited for a system in which the operating system kernel will be used to manage trace from multiple user processes.

A final difference between epoxie and pixie is that epoxie includes support for traced Mach 3.0 threads and for differentiating between instrumented and uninstrumented code. Pixie is limited to single-threaded users programs.

A number of tracing tools have followed these two early systems. Epoxie instruments and records trace for every basic block and every data reference. In QPT [9, 48] the frequency of such trace events is reduced through an analysis of program flow which allows some of the instrumentation sites to be eliminated. The analysis used in QPT reduces overhead due to tracing by 20-40%. In Shade [27] execution time for tracing experiments is improved by executing both the simulator and the workload in the same address space.

Epoxie was modified for this project to minimize the expansion of traced text. The text growth factor for epoxie is observed to range between 1.9 and 2.3, although actual growth depends on the length of basic blocks and the density of memory instructions. This compares very favorably with other instrumentation tools. Text commonly grows by a factor of five with the original epoxie. When used for address tracing, pixie commonly increases text size by a factor of six[2]. The text growth factor for QPT ranges from four to six [49]. It should be noted that, excepting the modified epoxie, minimal text growth was not a design objective for any of the these tools, and is only of importance when the additional virtual memory activity caused by text growth is an issue, as when monitoring system activity.

Other Measurement Systems

Researchers at Carnegie Mellon University used a combination of hardware and software support to implement event collection tools for analysis and tuning of parallel programs on the Digital Equipment Corporation M31 VAX multiprocessor [68]. This system used a hardware apparatus to collect and filter low-level events, modified microcode to monitor event sensors, and software tools for further analysis. They also explored tradeoffs for hardware, software, and hybrid methods for event collection and analysis.

Several other systems have used event counts or end-to-end measurements of execution time to detect and isolate performance problems in the memory system. These systems typically work at a level of abstraction familiar to the programmer, such as procedure boundaries and source-code lines. Although these tools are designed for detecting performance problems in user level code, they are haven't been adapted for use with operating system performance issues.

MTOOL [39] compares execution time of program segments to predicted time for a perfect memory system. A large difference between the predicted and the measured times implies a possible memory system performance problem. MTOOL was applied primarily to detecting memory bottlenecks in FORTRAN programs, and is not appropriate for understanding operating system behavior. MTOOL was adapted to work with shared-memory multiprocessor programs [40]. Another project, MemSpy [52], is based on the Tango [30] simulation and tracing system. Tango is designed for use with parallel applications and multiprocessor systems, and has not been applied to multiprogrammed uniprocessor workloads or measurements of operating system activity.

[2]The original `gcc` binary has 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 [49]. The modified epoxie grows `gcc` text to 1515520 bytes.

The measurements for this thesis concentrate on aggregate user and system behavior. In contrast, both MTOOL and MemSpy identify performance problems as specific segments of code (also data for MemSpy) within a user workload, as would be useful for tuning.

The PSpec system [64] allows the programmer to augment program source code with assertions about program performance. A runtime system then measures the validity of performance assertions, and provides notification when a performance assertion is violated.

2.2. Mach 3.0 and Ultrix

In this section we describe Mach 3.0 and Ultrix, to provide background for understanding the kernel tracing system as well as the cross-system comparisons in Chapters 4, 5, and 7.

The fundamental difference between Ultrix and Mach 3.0 is that Ultrix is a *monolithic* or *integrated* system, and Mach 3.0 is a *microkernel* or *kernelized* system. In a monolithic system, all system services are implemented in a single system context, the monolithic kernel. In a microkernel system such as Mach 3.0, primitive abstractions such as address spaces, task creation and destruction, and communication are implemented in the kernel, with higher-level systems services implemented in a separate protection domain as a server. Many current operating system text books discuss microkernel and monolithic kernel design. (See [17, 73, 77].)

The Mach 3.0 microkernel exports a small number of orthogonal abstractions including interprocess communication (IPC), threads, and virtual memory. Higher-level operating system services are implemented in a user-level process called the UNIX server. A program running on Mach 3.0 contacts the UNIX server through the Mach kernel's IPC interface [35], together with a user-level *transparent emulation library*, which is a shared library that is loaded into the address space of every process. The microkernel reflects UNIX system calls back to the calling program's emulation library, which converts the calls into RPCs to the UNIX server. Simple UNIX services such as `getpid()` and signal masking are handled within the emulation library.

Another difference between Ultrix and Mach 3.0 is in their UNIX implementations. Both Mach 3.0 and Ultrix support the 4.3 BSD Unix application programer interface

(API). This makes it possible to do a cross-system comparison using the same user executables. Although both systems support the same API, the implementation of UNIX primitives is not the same. Ultrix implements UNIX directly, while Mach 3.0 implements UNIX in terms of Mach 3.0 primitives. This extra layer of software abstraction leads to increased system overhead in Mach 3.0.

Mach 3.0 and Ultrix are further distinguished by differences in functionality between the systems. An example is in the virtual memory implementations. Ultrix virtual memory is derived from the original BSD abstractions [7], and is relatively machine-dependent. Mach 3.0 uses a more flexible and aggressive virtual memory system which exports a user-level interface and which is partitioned into a machine-dependent and a machine-independent layer [67]. This extended functionality is not required by standard UNIX workloads such as those used in the cross-system comparison.

Mach 3.0 is written to be largely machine-independent and portable. It runs on a large variety of current uniprocessor and multiprocessor platforms. This distinguishes it from Ultrix, which runs on DEC VAX computers and DECstation computers only. In Mach 3.0, all machine-dependent code is isolated by files and accessed through a procedure-call or constant-definition interface. In Ultrix, machine-dependent and machine-independent code is mixed within files, with the C preprocessor used extensively to isolate machine-dependent sequences.

For the cross-system comparisons, we have attempted to eliminate obvious superficial differences between the two systems. Both systems are compiled with the same compiler and at the same optimization level[3]. Both systems use a large file buffer cache (12 megabytes). Experiments on both systems were run in single user mode. The same scripts were used on both systems to run traced workloads.

[3]See Appendix B for details on software configuration.

2.2.1. Similarities between Mach 3.0 and Ultrix

Although the cross-system comparison between Mach 3.0 and Ultrix is motivated by the structural differences between the two systems, there are also substantial similarities between the two systems, and these similarities are the foundation for the comparison. To understand these similarities, we consider system activity in terms of the two external interfaces the operating system presents: the UNIX API, and the machine interface presented by the computer hardware. We will argue that activity which implements these external system interfaces is largely similar for Mach 3.0 and Ultrix. The remainder of activity, that which supports the linkage between these two external interfaces, is different between the two systems, and these differences are due to system structure.

The code base for the UNIX API is the same in both systems. Both UNIX implementations are derived from the BSD implementation of UNIX. This shared code base accounts for approximately 125000 lines of source code and 500K bytes of object code. This common code is in the UNIX server in Mach 3.0 and in the kernel in Ultrix. As both UNIX implementations are based on the same code, the activity to support this interface is comparable.

Three kinds of interactions dominate the operating system activity across the hardware interface: device management, trap handling, and virtual memory. Activity induced by device management and trap handling is similar for the two systems. Although VM related activity is different for the two systems, the differences are given consideration in the cross-system comparison.

The device drivers were redesigned for Mach 3.0 and are not the same as those for Ultrix. The I/O system in Mach 3.0 was designed to minimize machine and device dependent code, support user level device drivers, and to support location independence (i.e., devices located on another node of a hypercube), but without sacrificing performance [37]. The Ultrix and Mach 3.0 I/O implementations are different but their performance is comparable, as activity is determined largely by the requirements of device interaction and I/O related copy overhead, so this activity is comparable for the two systems.

The trap handlers were rewritten for Mach 3.0 and have slightly shorter code paths. Again, system behavior for this activity is largely determined by requirements of the hardware interface, and is comparable for the two systems.

Finally, the differences between the two VM implementations have already been noted, and their effects are isolated in our analysis. Some variations in VM activity (such as TLB faults) are due to system structure rather than VM implementation, and will also be considered in our analysis.

Overall, the similarities between Mach 3.0 and Ultrix establish that the observed differences in behavior for the two systems are due to structurally imposed activity and not arbitrary differences between the two systems. These similarities provide the foundation for the cross-system comparison.

2.3. Workloads

The comparison of Mach 3.0 and Ultrix in Chapters 4, 5, and 7 used a set of thirteen workloads, including workloads from the industry-standard SPEC benchmark suite and other well known UNIX utilities. Table 2-1 gives a brief description of the workloads used for the cross-system comparison. These workloads were selected to be representative of the kinds of workloads typically used for performance analysis on UNIX workstations. The shortest workload executes approximately ten million instructions. The longest workload executes over two billion instructions. Other workloads, specific to the X11 or TLB experiments and not used as a part of the cross-system comparison, are discussed in the related chapters in the context of those sets of experiments.

2.4. Conclusions

This work was motivated by two fundamental changes in computer systems: changing performance characteristics for uniprocessor computers, and changes in the structure of operating system implementations. The applicability of related prior work is limited by these two fundamental changes. This chapter has given an overview of prior work in system address tracing, and also of the changes in operating system structure that motivate the project. The information in this chapter provides the required historical background for the cross system comparison which appears in later chapters and is the main content of this research.

Workload	Description	Mach time	Ultrix time
sed	The UNIX stream editor run three times over the same 17K input file.	0.58	0.57
egrep	The UNIX pattern search program run three times over a 27K input file.	2.05	1.94
yacc	The LR(1) parser-generator run on an 11K grammar.	1.75	1.82
gcc	The GNU C compiler (gcc) translating a 17K (preprocessed) source file into optimized Sun-3 assembly code.	3.70	4.20
compress	Data compression using Lempel-Ziv encoding. A 100K file is compressed then uncompressed.	1.38	1.33
ab	The Andrew Benchmark with *gcc*. The assembler was not traced.	112.18	98.96
espresso	A program that minimizes boolean function run on a 30K input file.	6.23	6.46
lisp	The 8-queens problem solved in LISP.	56.46	54.97
eqntott	A program that converts boolean equations to truth tables using a 1390 byte input file.	66.05	65.85
fpppp	A program that does quantum chemistry analysis. This program is written in Fortran.	25.20	16.78
doduc	Monte-Carlo simulation of the time evolution of a nuclear reactor component described by 8K input file. This program is written in Fortran.	22.94	24.56
liv	The Livermore Loops benchmark.	1.24	1.22
tomcatv	A program that generates a vectorized mesh. This program is written in Fortran.	139.42	155.44

Table 2-1: Experimental workloads with execution times for a DECstation 5000/200.

Except where indicated, all programs are written in C. Execution times are in seconds, for runs with an uninstrumented binary and uninstrumented system. The bottom four workloads are floating-point intensive. None of the programs have been reordered or tuned for the underlying memory system.

Chapter 3
The Tracing System

The research for this thesis involved creating address tracing systems that allow complete and accurate measurements of activity on a computer system, then applying the tracing systems in three sets of experiments. This chapter explains the address tracing methodology, as well as some implementation details related to the experimental system. The first section describes some fundamentals of the software instrumentation technique used in this research. Section 3.2 discusses complexities involved in collecting address traces of operating system activity. Section 3.3 gives details about the memory system simulator used for the experiments. Section 3.4 describes techniques used to maintain the quality of trace data. Section 3.5 discusses methods used to evaluate and improve the quality of the address trace data. The chapter closes with a subjective evaluation of Mach 3.0 and Ultrix, based on experience from implementing the tracing system.

3.1. The Address Tracing System

The design of the tracing systems used in this study is based on earlier work at DEC Western Research Laboratory [16]. Important properties of the WRL design include:

- **Instrumented binaries:** Object code is rewritten such that address trace is generated as a side effect of program execution.
- **System Trace:** The ability to collect system trace was an original criteria in the design of the tracing system, and impacts many aspects of system design.
- **Multi-task traces:** By coordinating trace collection through the kernel, accurate interleaving of trace from multiple user and system contexts is possible.
- **Very Long Traces:** Trace is consumed *on the fly*, rather than being stored to permanent media. In this way arbitrarily long traces can be analyzed.

Figure 3-1 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. In addition to supporting user-level activity, the kernel is also involved in controlling the tracing system. Kernel activity that occurs on behalf of the tracing system is not traced.

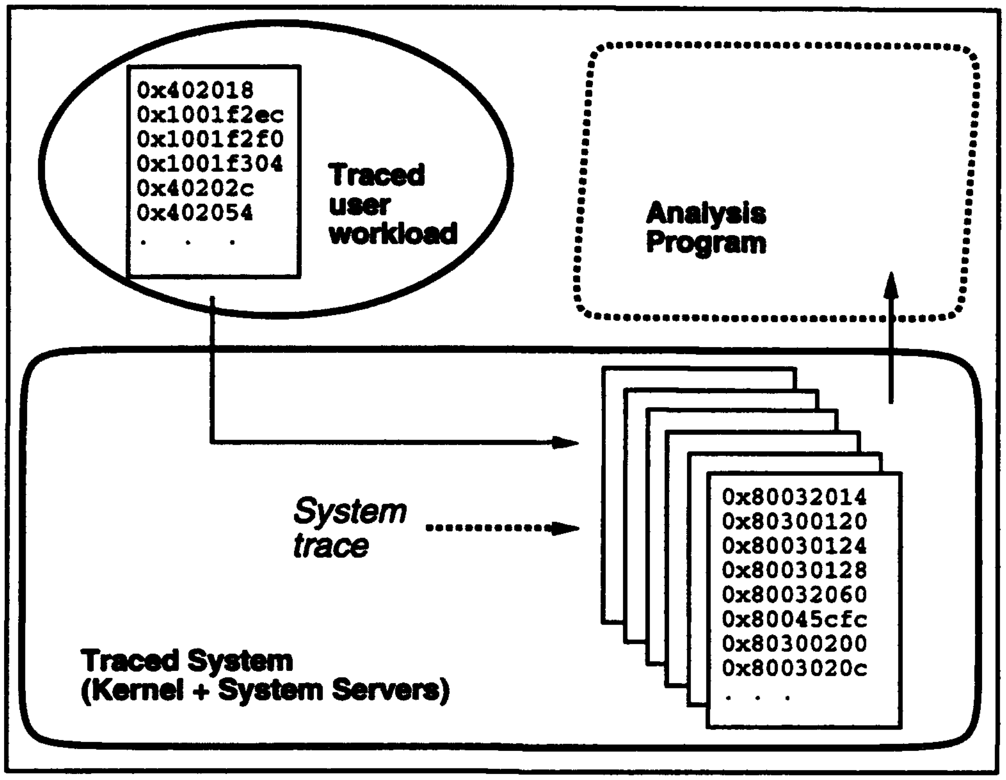

Figure 3-1: Overview of the tracing system.

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 user trace from per-process buffers when they become full, the kernel copies available user trace into the in-kernel buffer each time the kernel is activated. As every user-level context switch requires an invocation of the kernel to manage process state, the interleaving of trace from multiple user-level processes and the kernel is preserved.

A certain number of kernel modifications were required to support user tracing. Although required by the design of the tracing system they are independent of the generation of kernel trace. We modified the kernel memory initialization to allocate an in-kernel trace buffer statically at boot time. We modified exception handlers to copy trace from per-process buffers into the in-kernel buffer whenever traced user processes are interrupted. We added a mechanism for the analysis program to extract trace from the in-kernel buffer. In Ultrix, memory is accessed through a file abstraction, which is implemented similarly to the `/dev/kmem` abstraction of BSD UNIX[4]. In Mach 3.0, the in-kernel buffer is mapped into the virtual address space of the analysis program.

We added a kernel call in both systems to provide a mechanism for user-level analysis programs to control tracing. We modified process creation to initialize tracing data structures. We modified the scheduler to ensure that traced processes are inactive during trace analysis.

3.1.1. Epoxie

The tracing system uses the *epoxie* instrumentation tool [85]. Epoxie rewrites executable files, augmenting the original program text with instrumentation instructions.[5] We extended the original epoxie tool for this project to support kernel tracing, tracing of threaded address spaces, and to reduce the growth of instrumented program text.

Epoxie inserts trace-collecting code at the beginning of each basic block and before every memory instruction of the original program text. Figure 3-1 shows an example of a code sequence before and after instrumentation.

Each basic block is preceded by a three instruction sequence, as in instruction `i'+0`..`i'+2`. The `jal` instruction at `i'+1` is a call to a basic block trace routine, which will store the basic block address into the trace buffer. In actuality, `bbtrace` records the return address saved by the `jal` instruction, that is, the address of instruction `i'+3`. When the time comes for the trace to be fed to a simulator, the trace parsing library will

[4]`/dev/kmem` is used by UNIX utilities such as `ps` [80] to access kernel data.

[5]Address tracing is one of several types of instrumentation that tools like epoxie can do. Any reference in this document to instrumentation or instrumented binaries implies instrumentation for address tracing.

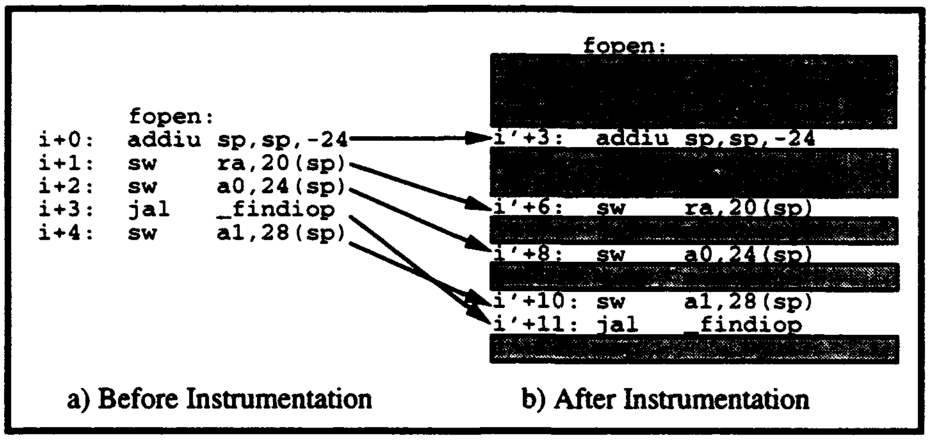

Figure 3-2: Instrumentation by epoxie

use static information recorded by epoxie to map this address to an address from the original (uninstrumented) binary.

The `jal` instruction destroys the return address register (`ra`), so it must be saved in the trace bookkeeping area (by instruction `i'+0`) before `bbtrace` is called. `bbtrace` and `memtrace` restore the contents of `ra` before they return. The delay slot of the `jal bbtrace` (instruction `i'+2`) contains a special no-op (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. A basic block generates one word of trace for the basic block address, and one word of trace for the address referenced by each memory instruction in the basic block. `bbtrace` uses the immediate field of this no-op to determine if there is enough room in the trace buffer for trace from this basic block to be stored.

The tracing system reserves three registers for its own use, which are referred to symbolically as `xreg1`, `xreg2`, and `xreg3`. The real values of these *stolen registers* are kept in a trace bookkeeping area, with `xreg3` used to store the address of the bookkeeping area. When a stolen register is used in the original code:

```
i:      addiu   xregX, xregX, -28384
```

the relevant instruction is replaced with a code sequence to use the shadowed value:

```
i+0:    lw      xreg1, XREGX(xreg3)
i+1:    nop
i+2:    addiu   xreg1, xreg1, -28384
i+3:    sw      xreg1, XREGX(xreg3)
```

In the instrumentation process, memory instructions are typically expanded into a two instruction sequence: a `jal memtrace` with the memory instruction in the delay slot. For example, instruction `i+2` from Figure 3-1:

```
i+2:     [fopen.c:  51]    sw     a0,24(sp)
```

becomes, after instrumentation

```
i'+7:    [fopen.c:  51]    jal    memtrace
i'+8:    [fopen.c:  51]    sw     a0,24(sp)
i'+9:    . . .
```

When `memtrace` is called, the register `ra` contains the address of instruction `i'+9`. `memtrace` partially decodes the instruction at `i'+8`, using the base register and immediate field to compute the address to be recorded in the trace.

In the presence of certain hazard conditions, it is not possible to put the actual memory instruction in the branch delay slot. For example, instruction `i+0`, below, overwrites its own base register:

```
i+0:     lw       t1, 84(t1)        /* reads and writes t1 */
```

If the load instruction at `i+0` were in the delay slot of a call to `memtrace`, the value in `t1` would be destroyed by the load before `memtrace` had a chance to compute the address for the trace. In such a case, a no-op (add to register zero) is inserted as instruction `i+1`, with register and immediate fields identical to those of the corresponding memory instruction. In this way `memtrace` can record the trace address before the load. The memory instruction is executed when `memtrace` returns. Similar hazard conditions occur when `ra` is a base or destination register in a memory instruction, or when a memory instruction uses a stolen register, and are handled similarly.

Recall that the shadow copy of `ra` is updated at the beginning of each basic block. When `ra` is modified within a basic block, the new value must be saved before the next `jal memtrace`. If this were the case for instruction `i+0` above, it would be expanded after instrumentation into the following sequence:

```
i'+0:    sw       ra, RADISP(xreg3)
i'+1:    jal      memtrace
i'+2:    addiu    zero, t1, 84      /* nop */
i'+3:    lw       t1, 84(t1)        /* reads and writes t1 */
```

The addition of instrumentation code increases text segment size. We made two major modifications to epoxie to reduce text expansion:

- Using a jump to a pc-relative address (a `jal` instruction) rather than a jump to an address contained in a register (a `jalr` instruction) reduced the number of instructions required for subroutine call sequence to a tracing routine.
- Data reference address extraction was made more compact. The original epoxie generated a sequence of instructions at the site of each load and store, computing the address to be traced before the call to the data reference tracing routine. In the modified epoxie, the address to be traced is computed by the data trace routine, which partially decodes the corresponding memory instruction and reads register contents to compute the data address. The new method executes more instructions for each data address traced, but the locality of those instructions is far better, with tens of instructions in a subroutine rather than several instructions inline for every memory reference. Because of this improvement in locality, the time to execute instrumented code improved, even though more instructions are executed.

With the original epoxie, the expansion factor for traced program text was approximately five. In the modified epoxie the expansion factor is about two. Note that the expansion of traced text does not affect the trace addresses generated, as text addresses that appear in the trace correspond to the original (as opposed to the instrumented) binary. The motivation for these modifications was to minimize the additional I/O and VM behavior that occurs as a result of text growth. I/O effects, VM effects, and other side-effects of tracing are discussed in Section 3.4.

3.2. Tracing the Kernel

The tracing system was conceived with complete system tracing as a goal, and features to facilitate such tracing are fundamental to its design. An example is the trace format. 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 makes it possible to preserve the interleaving of trace from various sources. For tracing tools such as pixie that manage trace at user level, preserving this interleaving is much more complicated.

Several peculiarities of operating system kernels make instrumentation a substantially different problem from instrumenting user level code. The foremost is the presence of uninstrumented code. Certain parts of the kernel are not instrumented by epoxie, either because they are part of the tracing system and should not be traced, or because they are too delicate to be rewritten by a mechanical tool. This is a problem because in general this code will use (and hence destroy the contents of) the registers reserved for epoxie. For this reason, stolen registers must be saved and restored during transfers of control between instrumented and uninstrumented code. Aside from kernel activity on behalf of the tracing system, all kernel activity is traced. Routines which are too delicate to be instrumented by epoxie were instrumented manually. Some code which is only executed at system boot time or after an unrecoverable system error is not instrumented.

A second problem is the need to manage the tracing system. Traced applications are serviced by the operating system kernel when their per-process trace buffers become full. The kernel itself generates trace, which is merged with user trace into a large in-kernel buffer. When the in-kernel buffer becomes full, the kernel must service itself. It must turn off kernel tracing, suspend traced user processes, and schedule the trace analysis program. Implementation of the software kernel tracing system required that this functionality be implemented in the kernel so as not to disturb the trace generated by normal system activity.

A third problem is that of 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 suspended for an interrupt or 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.

3.2.1. Implementation details for the two systems

We used epoxie to instrument both Ultrix and Mach 3.0. Epoxie operates on binaries *after* compilation, so registers reserved for tracing had to be "stolen," as described above. The necessity of register stealing complicated the implementation of the tracing system, creating additional overhead in trace-system state maintenance.

Epoxie generates static information describing each basic block (number of instructions, position of loads and stores). This information is used during trace analysis to determine the correct interleaving of instruction and data memory references. The handling of basic block records by epoxie was modified in preparation for system tracing. In prior versions of epoxie, basic block records were written into the trace along with the traced addresses. In the tool used for this project, only the basic block address is written. A lookup table is then used in the trace parsing library to find static information for a given basic block address. One advantage of this technique is that 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-case behavior 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 call a procedure which implements special-case handling of data in the trace. 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 of the idle-loop.

The Mach 3.0 virtual memory interface [67] 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 program's address space, eliminating copying and buffering of trace data.

Another use of Mach 3.0 virtual memory primitives is the dynamic allocation of per-process trace pages. In the Ultrix system, a flag is set in the executable image to indicate that a program is traced. This flag is used to identify traced binaries when a program is loaded. Processes flagged as traced 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 user references to the per-process trace pages. This feature in Mach 3.0 is particularly important for tracing of multiple 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.

Mach 3.0 supports two kinds of threads: Mach threads, implemented in the kernel, and user-level threads [28], implemented by a user-level library. User-level threads do not require special handing by the tracing system. Voluntary transfers of control between user-level threads occur at basic block boundaries and use standard mechanisms that do not require special support. Involuntary transfers of control imply an invocation of the kernel and a change of Mach threads, so trace state is managed by the mechanisms for Mach threads.

3.3. The DECstation 5000/200 Memory System

For many of the experimental results in this thesis, traces from the Mach 3.0 and Ultrix tracing systems were fed into a program that simulated the complete DECstation 5000/200 memory system. Many previous trace-based studies concentrate on the impact of cache performance in isolation from the rest of the memory system. They vary cache parameters such as associativity and cache size, while ignoring the impact of interactions between memory system components. Simulation of a cache in isolation can be useful for a memory system designer, but, as the cache is only one of several components in modern memory systems, it gives limited insight into memory system behavior as a whole, or the impact of memory system behavior on overall performance.

Parameters for the DECstation 5000/200 memory system simulation appear in Table 3-1. There are two main reasons for selecting this memory system. First, it is fairly conventional, with no unusual features (such as a very small TLB or virtually indexed cache) that would reduce the generality of the results. Second, Mach 3.0 and Ultrix both run on the DECstation 5000/200, permitting a comparison of simulation results with observed system behavior.

Of the memory system parameters listed in Table 3-1, one of them, virtual to physical page mapping, is determined not by hardware implementation but by policy implemented in the operating system. An address trace obtained through software methods contains *virtual* addresses, so a trace-based simulation of a large physical cache must use some policy for virtual-to-physical address translation. As the page mapping policy determines how pages overlap in the physical cache, it can have significant impact on cache behavior [22, 47]. Because this policy is orthogonal to operating system structure and because it has a negligible impact on the implementation, we have chosen to isolate it from other aspects of system design by implementing page mapping policy in the memory system simulator, and by using the same policy for both operating systems. As

instruction cache: 64 KB, direct-mapped, physical, 16 byte line, 13 cycle miss penalty.

data cache: 64 KB, direct-mapped, physical, 4 byte line, write allocate, 15 cycle read miss penalty, read miss fetches 16 aligned bytes.

write buffer: six entries, writes complete in five cycles; page-mode writes can be issued one per cycle. CPU reads have priority for memory access, but wait for writes that have already been issued. 4 KB page size for page-mode writes. Partial writes complete in 11 cycles.

translation buffer: 64 entries, 56 random/8 wired entries, trap to software on TLB miss. Each TLB entry maps a 4 KB page.

page mapping: Deterministic. The physical page used to back a given virtual page is determined by the virtual page number and its address space identifier. The deterministic strategy prevents conflicts within any 64 KB (cache size) range of virtual addresses.

kernel memory: All kernel text and most kernel data is in unmapped, cached physical memory.

Table 3-1: Memory system simulation parameters.

an alternative to page mapping policies implemented in the simulator, the Ultrix and Mach 3.0 tracing systems also provide a mechanism for dynamically extracting the page-map from the running system. The impact of page mapping policy is investigated in Chapters 4 and 5.

3.4. Ensuring Trace Quality

Instrumenting the system involves substantial modifications to all 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.

3.4.1. Avoiding Trace Distortion

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

Memory Dilation

Program text 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 avoid paging as a simplification of memory system behavior, as this research is meant to focus on components in the memory hierarchy between the CPU and memory caches. The behavior of these parts of the memory hierarchy becomes irrelevant when significant paging activity is present.

Memory dilation also affects TLB behavior. The model for TLB miss handling in these experiments is based on that of the MIPS R3000 processor used in the DECstation [46]. The address space of the DECstation is divided into four segments, two mapped and two unmapped. All kernel text and most kernel data is referenced through the unmapped segments, so 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. These are handled through the general exception mechanism, which is much slower (several hundred instructions). Fortunately, KTLB misses are less frequent.

Instrumentation tends to double the number of active user text pages. 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 simulator-generated misses to synthesize the activity of the UTLB miss handler.

The mapped kernel segment 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. The largest user binary (the Mach Unix server) has less than two megabytes of text after instrumentation, the number of page table pages it requires does not change, and the behavior of the KTLB miss handler is not affected.

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, time dilation 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.

Asynchronous I/O events do not cause user programs to wait, so they have minimal impact on execution time for user programs. Synchronous I/O operations have a greater effect on running time, as they cause the user program to wait for the completion of a disk request. 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 the latencies of synchronous disk operations using a count of the number of instructions executed while waiting in the system idle-loop. We estimate the corresponding latency for an untraced system by scaling idle activity in the traced system by a factor of fifteen. This approximation is rough but adequate for our purposes. I/O delays are of little importance in memory system behavior, as the memory system behavior of the idle loop is uninteresting and has little impact on other activity in the machine.

Scheduler policy is also affected by time dilation, as scheduler behavior for multitasking workloads depends on the dynamic patterns of overlap between I/O latencies and clock events. Scheduler policy is an issue we have chosen not to address. Instead, we concentrate on single-process and client-server workloads. For these workloads, all context switches are determined by client-server relationships, and scheduler policy is irrelevant. Accurate traces of timesharing workloads would require accurate scaling of I/O delays and adjustment of scheduler policy to replicate untraced behavior. It should be possible to reduce the distortion of scheduler behavior in the traced system, although perfect reproduction of behavior is not a practical goal. Given current trends away from multi-user timesharing, the restriction of client-server workloads does not significantly limit this research.

3.4.2. Defensive Tracing

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

The tracing system was further tuned and corrected by looking for anomalies in measured behavior. In this section we discuss redundancy and error modes in the tracing system which are used to detect certain types of errors. In the next section we discuss the end-to-end measurements used to evaluate the quality of address trace data, in which trace driven simulation was used to predict behavior of an uninstrumented system.

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 has an important role in controlling the state of the tracing system. In implementing the tracing system, it was important to manage transitions between these different types of code to avoid omissions or distortions that would cause the address trace to differ from behavior in the untraced system. Several 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 references to kernel segment addresses are from the kernel.
- 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 pos-

sible to identify and eliminate anomalous activity such as trace generated by code specific to the tracing system.

Each transition in the tracing system from trace-generation to trace-analysis creates a situation in which distortion can be introduced into the address trace, as some amount of system activity can be ignored or distorted. 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 avoid such loss of trace was to be sure that such events 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 to about two seconds of continuous execution, long enough so that the distortion from trace phase transitions is not significant.

3.5. Validation of Methods

This section discusses some measurements and observations that demonstrate the quality of the behavioral model provided by the tracing/simulation system. First we discuss two end-to-end measures [71] used to evaluate the quality of measurements from the experimental system to real system behavior: prediction of execution time and prediction of TLB miss behavior. Then we give some examples of unexpected behavior revealed by the experiments, as evidence of the power of the measurement system for revealing counter-intuitive behavior. Overall, these measurements demonstrate that the experimental system provides an accurate and detailed model of system behavior.

3.5.1. Program Execution Time

Latency in program execution can be reduced to the sum of latencies of low-level events such as instruction execute cycles and cache misses. The experimental system can be used to obtain counts of these events. The event counts and associated latencies can be used to predict the execution time of a given workload. We compare the execution time as predicted by the experimental system to the execution time as measured with an accurate timer, as an indication of the ability of the experimental system to model the behavior of the uninstrumented system.

Table 3-2 compares program execution time as measured by an accurate timer with execution times predicted from by the experimental system for trace-driven simulation of the DECstation 5000/200 memory system.

	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 3-2: 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 [75].

The measured times in this table are different from those in Table 2-1. For the runs in this table, the buffer cache was warmed by reading the program executable from disk before running the workload, to minimize the activity associated with the loading of program text. Also, a revision 3.0 of the floating point unit was used. This causes significant variation in running times for *fpppp* and *doduc*. Page mapping policy also contributes to large variations in execution times between runs on Mach 3.0.

The predicted times in Figure 3-2 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 [75] 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 adjusted 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 coarse 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.
    - Some system activity is missed when tracing is interrupted during a disk request.
    - Tracing changes the behavior of disk read-ahead. Some read-ahead requests complete in the traced system but do not 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 cache misses and write buffer cycles 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 3-3 shows percent error for predictions of execution time for twelve workloads

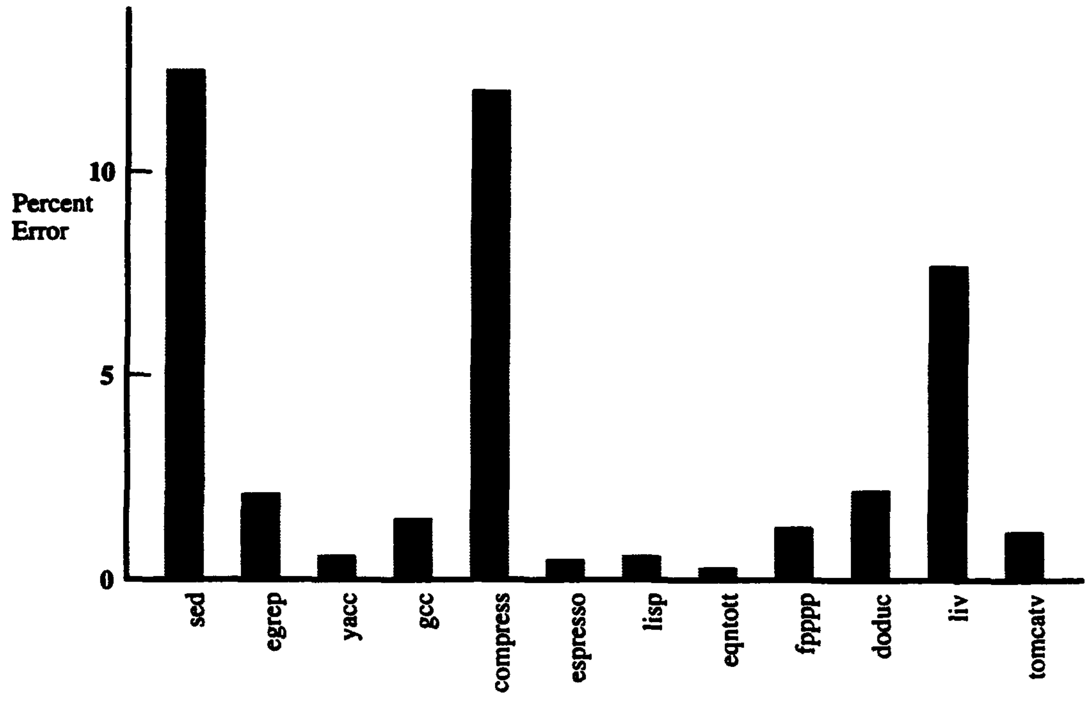

Figure 3-3: Error in predicted execution times for Ultrix.

Three of the workloads, *sed, compress*, and *liv* show large errors in predicted execution time, due to known inaccuracies in the tracing system. See the text for a complete discussion.

running under Ultrix[6]. Predictions for most of the workloads are quite good. Three of the workloads have errors greater than five percent. The explanation for these errors gives interesting insights 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, 150K bytes, but its execution time is only 1.32 seconds. The prediction error is mostly due to disk read-ahead phenomena, where read-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 trace/simulation system and idle time measured by the timing facility in the c-shell [81] confirms that the simulator does under-estimate idle activity.

[6]Because of the large variability of running time induced by the Mach 3.0 random page mapping policy [47], we do not present error figures for Mach 3.0.

- *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 considered in this study. 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 no impact on the sequence 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. Overall, the good estimates of running time for most of the workloads demonstrates that the address trace collection is accurate. The measurements provide strong empirical evidence that errors in predicted execution time are due to limitations in the system model and not errors in the address traces.

3.5.2. User TLB Miss Count

User segment TLB miss activity provides another opportunity for comparing activity predicted by the experimental system to activity measured in an uninstrumented system, again as an indication of the ability of the experimental system for predicting program behavior. The DECstation handles user-segment TLB misses using a small miss handling routine in software. This miss-handling routine can be modified to collect user-segment TLB miss counts without otherwise disturbing activity on the machine. In Table 3-3 we compare miss counts from the modified miss handling routine to the miss counts predicted by the memory system simulation.

Although the miss rates predicted by the simulator correlate within an order of magnitude, the percent error is sometimes large. One source of error in the TLB miss predictions is the random TLB management policy used in the DECstation, which introduces variability in miss activity for different executions of the same workload. Another source

	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 3-3: TLB misses, measured and predicted.

of error 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. The simulator doesn't know about these writes, so the corresponding 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 3.0.

Given the type of activity and its impact on performance, the error in predicted TLB behavior 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.

3.5.3. Explaining Anomalous Behavior

In debugging the tracing system, a procedure used repeatedly was to identify anomalous behavior indicated by simulator output, and use it to find bugs 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. This section documents some of the more interesting findings.

Uncached Instructions in Mach 3.0

Early experiments showed that, for a given workload, Mach 3.0 executed many more uncached memory reads than Ultrix. Uncached reads normally correspond to I/O operations. I/O devices write directly to memory, so the system must read through uncached memory in order to avoid stale data that might still be in the cache. As both systems were running identical workloads, it was expected that the amount of I/O for both systems, and hence the number of uncached memory reads, would be the same. A more careful look at output from the simulator revealed that the uncached reads were due to instruction reads, not data reads as I/O operations would have generated. The precise addresses of the uncached instruction reads revealed a performance bug in Mach 3.0. The Mach instruction cache flush routine was executing an inner loop from uncached memory when it should have been reading instructions through the cache.

Data Cache Behavior in tomcatv

Table 2-1 shows that *tomcatv*, a scientific computation written in Fortran, runs faster on Mach than on Ultrix by fourteen seconds, a difference of about ten percent. As the simulator indicated that only 1% or less of total execution time was spent in the system, it seemed impossible that system activity could account for a ten percent change in running time.

The first hypothesis was that the experimental system was failing to record some Ultrix instructions, but no such bug could be found. Then we looked at data cache behavior. Using the page table information recorded in the address trace, we discovered that Ultrix was assigning virtual pages to physical pages in a very regular way, and in such a way that various matrices used in the computation were aligned in the cache. For *tomcatv*, the deterministic policy used in Ultrix to assign virtual to physical pages causes many more conflict misses than the policy used in Mach. Hence system policy and not system structure was the cause of the difference in execution time. See Section 4.3.7 for more discussion on virtual to physical page mappings.

I/O time in Ultrix

As can be seen from Table 2-1, many of the workloads run faster under Mach 3.0 than under Ultrix — even system intensive workloads such as *gcc*. This seemed counter-intuitive, as system overhead from instruction execution and memory system penalties is higher in the microkernel system than in the monolithic system (as shown in Chapter 4). Some of the performance penalty is due to page mapping policy, as described for *tomcatv* above. I/O intensive workloads get a further advantage under Mach 3.0 due to the conservative disk management policy in Ultrix. Ultrix writes file system meta-data (such as last-access times) synchronously, whereas Mach 3.0 uses asynchronous updates. Another difference is that Ultrix preloads complete program text when a program is executed, but Mach 3.0 brings in text pages as needed using the page fault mechanism. Consequently, for I/O intensive programs, the Ultrix policies result in more synchronous I/O activity hence more time in the idle loop. This difference in I/O behavior is important in explaining observed performance differences between Ultrix and Mach.

Floating Point Traps in *fpppp*

Measurements of the *fpppp* workload showed that most system activity was due to floating point exceptions. This behavior was unexpected, as no floating point exceptions were thought to occur.[7] After re-checking the experimental system and trying a different Fortran compiler, we eventually discovered that the exceptions were dependent on the version number of the floating point hardware. The FPU hardware in the traced machine generated software traps for certain conditions that were handled without a trap by later versions of the chip.

3.6. Experience with Mach 3.0 and Ultrix

Much of this document concerns a quantitative comparisons of Ultrix and Mach 3.0. As this project involved functionally similar modifications in both systems, a qualitative comparison is also possible.

Work on the traced Ultrix system began in the summer of 1992 and the tracing system was largely completed by December of 1993. Implementation of user-level tracing was the first step in this process, and was completed by August of 1992. Development of the traced Ultrix system required approximately eight man-months of work, and occurred concurrently with development of other parts of the tracing system.

The tracing system for Mach 3.0 was implemented during a two month effort, from mid-January to mid-March 1993. Support for user-level tracing was implemented first. Next the Mach kernel was instrumented, then the UNIX server, then the emulation library.

[7]Engineers at Digital Equipment Corporation claimed that no such exceptions occurred when they ran the workload on their systems [72].

There are several reasons why development of the instrumented system took significantly longer for Ultrix than for Mach. The tracing system for Ultrix was the first to be implemented. Experience gained during the Ultrix implementation helped to avoid problems when instrumenting Mach 3.0, and made bugs in Mach 3.0 easier find and correct. Additionally, the debugging of epoxie and other tools occurred during the instrumentation of the Ultrix system. Finally, the traced Mach system was developed at Carnegie Mellon University, and Mach expertise was readily available to aid in the design and debugging of the tracing system. This is in contrast to Ultrix, for which no expertise was available. Information about specific aspects of Ultrix was much more difficult to obtain.

The above considerations are independent of actual differences between the two systems. In the following we describe some of the differences between Ultrix and Mach 3.0 that were relevant to the creation of the tracing systems.

Additional system contexts

The structure of the two systems, monolithic vs. micro-kernel, influenced the ease with which modifications could be made to the two systems. Having three traced system contexts in Mach 3.0 (kernel, UNIX server and emulator) as compared to one in Ultrix (the kernel) added new problems and complexity to the development process due to the additional configuration files and executable images involved in the creation of the system. The decomposition of Mach 3.0 had minimal impact on the design of the tracing system.

Debugging

Overall, Mach 3.0 provides superior support for system debugging, with an interactive debugger built into the kernel. The Mach kernel debugger was useful for debugging the traced UNIX server and traced emulation library. It was less useful for the traced Mach kernel, for several reasons. First, many of the difficult bugs occurred early in the boot sequence, too early for the debugger to be used. Secondly, the traced Ultrix kernel was developed without a kernel debugger, so debugging techniques developed for Ultrix did not make use of the interactive debugger. Lastly, tracing the kernel had the potential to interfere in significant ways with the debugger, and the expected benefits did not justify the potential extra effort required to debug the debugger. In the end, the debugger did work, and the code modifications required to make it work were straightforward.

Virtual Memory and Address Space Considerations

In Ultrix, the allocation of per-process trace pages was among the most difficult new system functionality to implement, and was the source of a number of difficult bugs. This implementation was less difficult in Mach 3.0, due in part to the carefully designed Mach virtual-memory interface. In Ultrix, the VM code was mostly undocumented, uses short identifier names that are meaningless to someone unfamiliar with the code, and is difficult to isolate from the rest of the system. With the Mach VM interface, it was possible to manage user trace pages using a small number of existing VM primitives, and with minimal changes to existing code. In Ultrix, new low-level operations were required to allocate and deallocate user trace pages. Overall, these differences made the Ultrix code harder to understand and harder to debug.

The refinement of the Mach VM system made it possible to support threaded user address spaces with minimal and highly localized modifications. Ultrix does not support multiple threads in a single address space, so Ultrix support for traced threads was not required. Similar changes in Ultrix would probably have been much more difficult.

Additional functionality in the Mach 3.0 VM interface made it possible to improve the implementation of the tracing system in several ways. The memory mapped device interface in Mach made it possible to map the shared kernel buffer into the address space of a user process. This eliminated the need to copy trace from the kernel into user space, as was necessary in Ultrix, and simplified the coding of performance-critical components of the tracing system.

Chapter 4

Mach 3.0 and Ultrix: The Impact of Structure

In this chapter we evaluate and compare the impact of structural differences between DEC Ultrix and Mach 3.0, with emphasis on memory system behavior. In Section 4.1 we discuss how the effects of the structural differences between Ultrix and Mach 3.0 were isolated from effects due to policy. In Section 4.2 we discuss the measured differences in system behavior and performance between Mach 3.0 and Ultrix. In Section 4.3 we evaluate the monolithic and microkernel implementations in the context of the assertions in Table 1-1. Finally, in Section 4.4 we summarize our results.

4.1. Isolating the Effects of Structure

The principle differences between the two systems were reviewed in Section 2.2. Here we summarize that section, stating differences as they relate to the results presented later in this chapter.

The most important distinction is that Ultrix has a monolithic structure and Mach 3.0 is implemented as a microkernel, with the UNIX API implemented as a user level UNIX server. This has impact on many aspects of system behavior, including traps, context switches, and TLB miss activity. Another relevant distinction is that Ultrix implements the abstractions such as UNIX process management directly with low level primitives, while Mach 3.0 implements these abstractions in a user-level server in terms of Mach primitives. A third relevant distinction is the VM implementation. The functionality of Mach 3.0 VM differs in several ways from that of Ultrix, and these differences impact the performance of the VM system.

For our comparison of Mach 3.0 and Ultrix structure we exclude two kinds of system policy. These policy differences were discussed in terms of specific workloads in Section 3.5.3. One is disk I/O policy. Disk write policy is different in the two systems, and this has significant impact on overall behavior. Ultrix attempts to maintain the disk in a consistent state, using synchronous, blocking writes when updating file-system meta-data. In Mach 3.0 disk consistency is compromised for improved performance by leaving meta-data in memory in the disk cache rather than writing it immediately to disk. The result is that for a given workload, Ultrix issues more synchronous disk requests than Mach, resulting in greater idle instruction counts and more disk delays. Further, program text under Mach is demand-paged, whereas under Ultrix it is loaded entirely at program startup, and this sometime leads to unnecessary disk reads in Ultrix. These differences are reflected primarily as time spent in the idle loop, waiting on the completion of synchronous I/O requests. As I/O policy is orthogonal to kernel architecture and to memory system behavior, we exclude idle-loop activity from the measurements and consider only non-idle events.

Another policy difference which we have isolated for our cross-system comparison is virtual-to-physical page mapping policy. Ultrix uses a deterministic page mapping policy similar to that used for our base memory system simulation (see Table 3-1). The page mapping policy used by Mach 3.0 is essentially random. These differences can have significant impact on memory system behavior. However, as the code implementing page mapping policy is concise and isolated, it is irrelevant to other aspects of system implementation and behavior. We use the same deterministic page mapping policy for both our Mach 3.0 and Ultrix simulations. We explore the impact of page mapping policy on memory system performance in Section 4.3.7. In Chapter 5 we consider the relative impacts of policy and structure on overall performance.

4.2. Comparative System Behavior

A summary of the trace results for each program is shown in Table 4-1, with aggregate results shown in Table 4-2. For Ultrix, all system activity is due to the kernel. For Mach, system behavior includes the kernel, the UNIX server, and the emulation library. For workloads that rely heavily on UNIX services, the combined Mach system components (microkernel, UNIX server, and emulation library) execute more instructions and generally require more data references than Ultrix.

	non-idle instructions				idle instructions		instruction cache misses				data cache reads				data cache read misses			
	Ultrix	%s	Mach	%s	Ultrix	Mach	Ultrix	%s	Mach	%s	Ultrix	%s	Mach	%s	Ultrix	%s	Mach	%s
sed	5704	24	7763	44	5876	1270	51	96	149	98	1834	32	2693	53	16	97	70	98
egrep	43277	4	45029	7	2495	914	43	93	140	98	9363	7	10057	14	32	91	71	97
yacc	32799	6	34539	10	13220	2809	69	89	166	96	7322	11	8027	19	48	50	93	71
gcc	29318	22	35939	36	63684	27027	485	42	999	71	9646	25	12754	43	120	44	318	71
compress	16896	19	19926	31	5555	2225	70	96	215	97	5459	23	6596	36	166	28	271	50
ab	869732	33	1198172	51	689324	247969	15612	52	28619	73	283994	42	398954	58	6658	79	11262	86
espresso	135385	2	137806	4	21601	8069	187	45	344	70	32568	3	33651	6	93	32	168	58
lisp	1288027	3	1276619	2	1005	0	222	61	2004	54	475185	3	473281	3	655	45	734	68
eqntott	1414369	1	1417868	1	10632	0	126	88	254	97	298601	2	300299	2	14328	3	14489	4
fpppp	265457	8	262998	7	17102	5667	4135	21	3735	19	141178	3	141485	4	131	27	177	47
doduc	321325	1	325351	2	18474	4983	6239	5	6292	7	122630	1	124410	3	550	9	612	21
liv	23008	3	23778	6	1585	639	21	93	72	98	8134	4	8458	7	17	88	30	96
tomcatv	2005703	1	2005590	1	10823	134	138	82	326	84	970227	0	971055	0	85451	0	85522	0

Table 4-1: Summary of trace results.

This table shows the number of non-idle memory system events, and the percentage due to system behavior for each program on both operating systems. Additionally, the table gives the number of idle instructions executed. All counts are in thousands.

	i-cache cycles	d-cache cycles	tlb cycles	wbuffer cycles
Ultrix user	0.07	0.08	0.00	0.02
Mach user	0.07	0.08	0.00	0.02
Ultrix system	0.43	0.23	0.00	0.05
Mach system	0.57	0.29	0.01	0.07

Table 4-2: Summary penalty cycles (per instruction).

These aggregate measurements are the average over the workloads from Table 2-1. They show system cycles per system instruction and user cycles per user instruction, and are intended to emphasize the difference between dynamic execution of system versus user code. This is in contrast to calculations of *MCPI* (Figure 4-1), for which total (system + user) instruction counts are used.

Memory cycles per instruction

We use our simulation results to calculate *memory cycles per instruction* (*MCPI*), which is the number of CPU stall cycles due to the memory system divided by the number of instructions executed. *MCPI* is one of several components of cycles per instruction (*CPI*), which is a metric commonly used to evaluate computer systems [42]. Other components of *CPI* (such as one cycle per instruction for instruction execution, interlocks during multiply, divide, and floating point operations, and no-ops inserted by the compiler for load and branch delays) remain relatively constant even as processor cycle time decreases. In contrast, *MCPI* is a function of the ratio of memory speed to processor speed, is less dependent on processor architecture, and will dominate overall *CPI* if current trends in processor and memory speed continue. As mentioned, we have excluded idle-loop activity from our *MCPI* calculations. The idle loop rarely misses in the cache so a system could achieve an artificially low *MCPI* by executing an arbitrarily large number of idle instructions.

The *MCPI* for each workload under Ultrix and Mach 3.0 is shown in Figure 4-1. Each bar is shaded to denote different *MCPI* components. System and user contributions are separated by a vertical bar. The figure shows that data and instruction cache misses in user and system mode are only partially responsible for the total *MCPI*. Other components include CPU write-stalls and kernel uncached memory reads. CPU write-stalls are reflected in Figure 4-1 in the *wbuffer* component, which shows the average per-instruction penalty from writes to a full write buffer as well as reads that stall pending the completion of a five-cycle write. A system uncached memory read occurs when the kernel accesses memory through the uncached segment [46] such as for I/O or device control. TLB misses do not appear explicitly in Figure 4-1. Their cost appears as additional instructions and data references, which are included in the total counts.

The *MCPI* components of the various programs reflect their internal behavior. The programs *sed, egrep, yacc, gcc*, and *compress* all have relatively high system *MCPI* components due to their greater reliance on the operating system, especially the file system. The *gcc* compiler, while run on a relatively small input file, has a large program text and requires more system activity during program loading. The scientific workloads (*fpppp, liv, doduc, tomcatv*) are dominated by user activity, as shown by their small system *MCPI* component.

4.2.1. Breakdown of System Activity

As shown by Table 4-1 and Figure 4-1, the most significant difference between Mach 3.0 and Ultrix is the number and cost of non-idle system instructions required to run an application. In this section we consider how different kinds of system activity contribute to dynamic instruction counts for both system.

In Figure 4-2 we separate system overheads by 11 major activities to facilitate the comparison of Ultrix and Mach 3.0. The components are:

- *trap* (system call, interrupt, and exception trap handling), *UTLB* (user TLB miss), *KTLB* (kernel TLB miss), *VM-md* (machine-dependent virtual memory), *VM-mi* (the Mach machine independent virtual memory layer), *Block Ops* (block memory moves and zeroes), *UNIX service* (the remaining routines in the Ultrix kernel and Mach UNIX server), *Microkernel* (the Mach

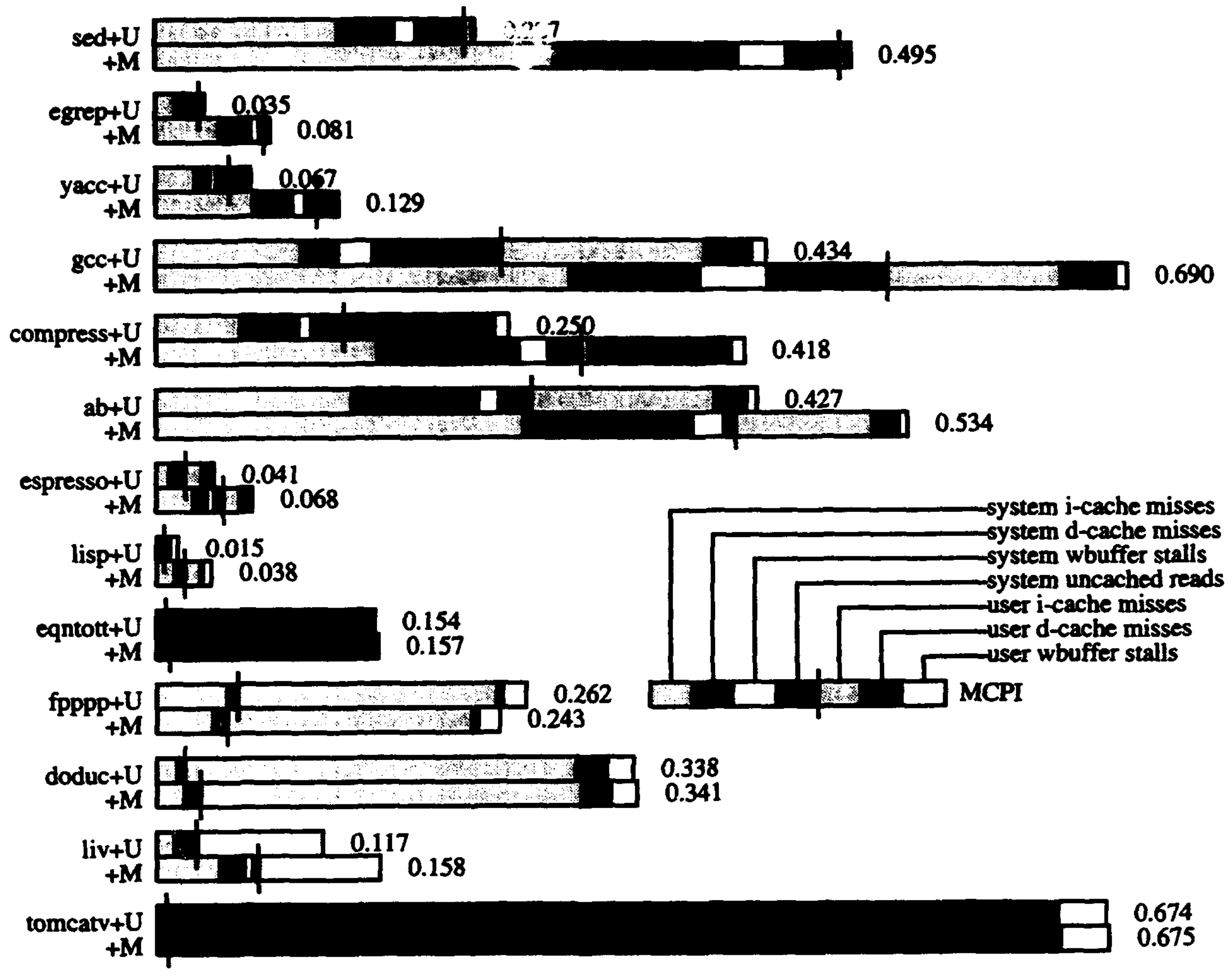

Figure 4-1: Baseline MCPI for Ultrix and Mach.

The top horizontal bar of each pair is for Ultrix (+U) and the bottom is for Mach (+M). Components to the left of the vertical line are due to system activity and those to the right are due to user activity. The number at the right of each bar is the *MCPI* for that workload. Idle-loop activity is excluded from these measurements.

microkernel, including device management and scheduling), *IPC* (Mach inter-process communication), *Emulator* (the Mach transparent emulation library), and *S-MCPI* (system memory cycles per system instruction).

S-MCPI is an indication of the memory-system overhead of system activity[8]. Note that

[8]*S-MCPI* is computed as *system cycles / system instructions*. It differs from *MCPI* due to the system (as in Figure 4-1) in that *MCPI* shows overhead per instruction for total (user+system) instructions.

four of the activities (*Microkernel, Emulator, IPC, VM-mi*) occur only in Mach. *Block Ops* for Mach includes operations from both the Mach kernel and the UNIX server. The Ultrix instruction counts have been normalized to one for all workloads. The heights of the bars reflect *system*, but not *total* execution, overheads. The number at the top of each bar is system activity as a percentage of total non-idle cycles. The workload *doduc* demonstrates the importance of this number; although each system instruction is relatively expensive (about 2.6 cycles per instruction for Ultrix), the system is not very active (only 2.2% of total machine cycles). Therefore the overall impact of system activity is small.

Several characteristics of system behavior are worth noting from Figure 4-2. The overhead of Mach 3.0 IPC is responsible for a small portion of overall system overhead in terms of instructions executed. This suggests that microkernel optimizations focusing exclusively on optimizing critical paths in IPC [13, 34] could have a limited impact on overall system performance, confirming an earlier result [12]. The *UNIX service* category between Ultrix and Mach 3.0 cannot be compared directly. For Ultrix, *UNIX service* includes many machine-dependent services such as device management that are counted as part of the *Microkernel* category in Mach. The combined size of *Microkernel* and *UNIX service* components for Mach indicates the cost of providing UNIX services with a user-level server through Mach 3.0 kernel interfaces.

UNIX service in Ultrix also includes some activity appearing under the *Emulator* category for Mach 3.0. For example, *lisp* has a relatively high *UNIX service* component under Ultrix, but almost none under Mach. This is because *lisp* frequently modifies UNIX signal state to support garbage collection, and signal state is manipulated from within the Mach emulation library. Overall, the cost of microkernel (as opposed to monolithic) system structure is reflected in Figure 4-2 as higher instruction counts for Mach 3.0, with impact on all components of system activity.

The implementation of the UNIX API as a user-level server creates a memory-mapped system context in Mach 3.0. In Ultrix, the corresponding functionality is implemented in the unmapped kernel context. As seen in Figure 4-2, the mapped system context in Mach 3.0 induces increased *UTLB* and *KTLB* activity for Mach 3.0.

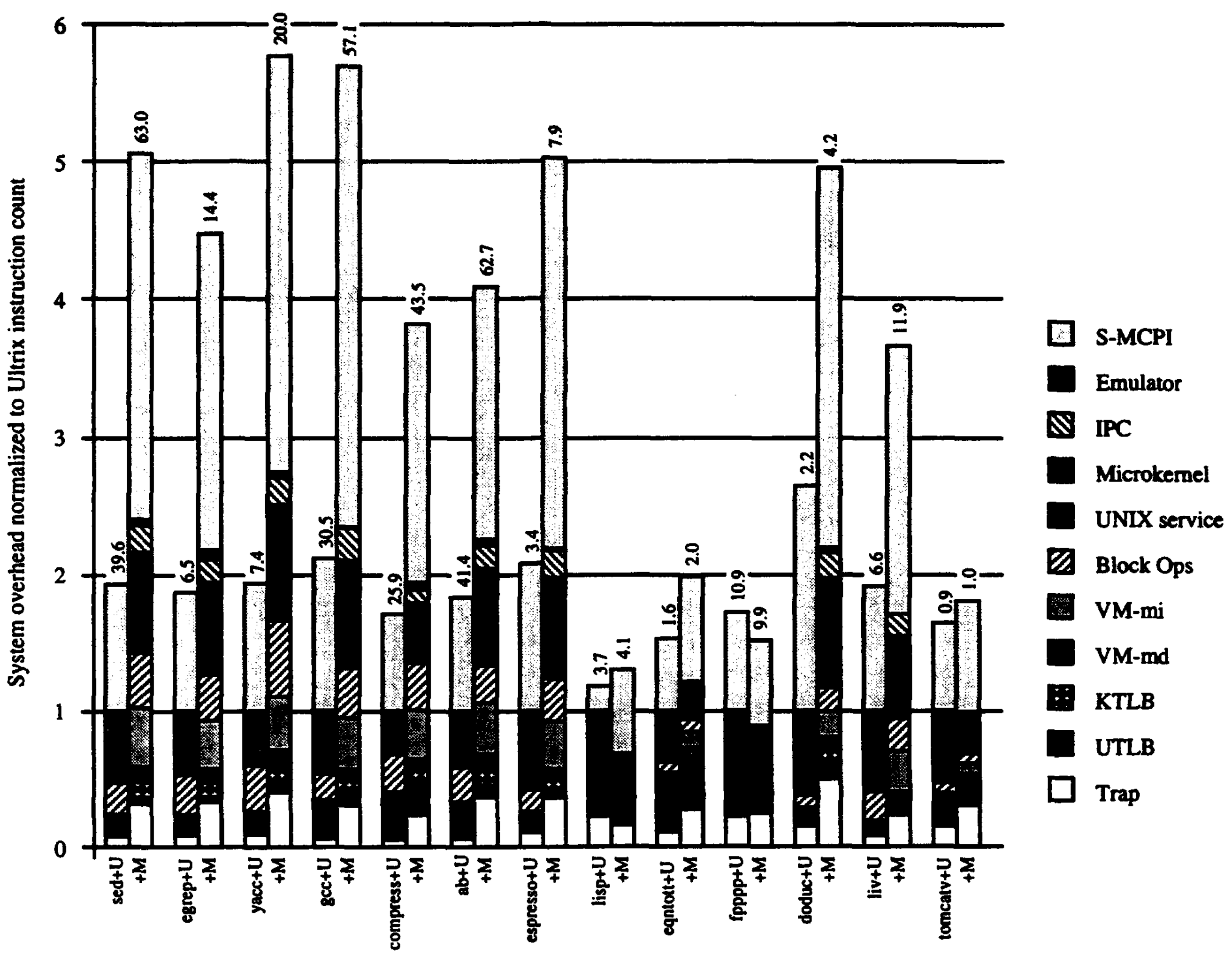

Figure 4-2: Relative system overheads for programs running on Ultrix and Mach.

See page 71 for a color version of this figure. This figure shows the relative system instruction and system memory overheads for programs running on Ultrix (+U) and Mach 3.0 (+M). Ultrix instruction counts are normalized to one. The top component of each bar is *S-MCPI*, which is an indication of the memory system overhead induced by system instructions. The number at the top of each bar is the percentage of total (instruction and memory) cycles that are due to the system. For programs where the system is responsible for a small percentage of total cycles, system overheads are relatively unimportant.

A further change introduced by the Mach 3.0 microkernel decomposition is that the buffer cache of disk data was moved from the kernel into the user-level UNIX server. Also, certain *Block Ops* routines in the UNIX server were implemented in C, rather than hand-coded assembler as is used in the Ultrix implementation. These changes are reflected in Figure 4-2 as the increased instruction counts for *Block Ops* and Mach 3.0 *UNIX service*.

The Mach virtual memory system executes more instructions than that of Ultrix. Ultrix VM is implemented as a single machine-dependent layer. The machine-independent VM layer in Mach 3.0 is generally more costly than either systems' machine-dependent layer. This comparison reflects in part the overhead required to support the additional functionality and portability of Mach virtual memory. Other differences in VM performance are due to differences in functionality. For example, Mach maintains dirty bits for all user data pages, whereas Ultrix assumes all data pages are dirty.

Overall, Figure 4-2 demonstrates two important points. First, there is no obvious flaw or glaring deficiency that explains the larger instruction counts for Mach 3.0 as compared to Ultrix. Mach 3.0 executes more instructions, and these instructions are distributed across many types of system activity. Second, system behavior varies widely depending on workload. Attempts to characterize user behavior with a single number or with a small number of workloads hide an enormous amount of information and are misleading.

There are two workloads for which system instruction counts are actually lower for Mach 3.0 than for Ultrix. For *lisp*, system activity is dominated by signal handling and maintaining signal state. Lower instruction counts for Mach 3.0 reflect the shorter code path in Mach for these activities. Similarly, system activity in *fpppp* is dominated by traps for floating point operations not implemented in hardware[9] The code path in Mach 3.0 for handling these traps is shorter. Lastly, system activity in workloads such as *tomcatv* is dominated by clock interrupts and TLB faults, with roughly equal overhead for both systems.

The memory system penalty for system instructions, reflected in Figure 4-2 as the *S-MCPI* category, is from one to three times greater for Mach than for Ultrix. The difference in the system *MCPI*, while sometimes small (Figure 4-1), can contribute substantially to overall system performance because of the large number of system instructions executed. We break down the memory system overheads for instruction and data caches by activity in Figures 4-3 and 4-4. Overall, memory system penalties from the different classes of activity are as would be expected given the dynamic instruction counts from

[9]This behavior is dependent on the version number of the floating point chip. Later versions of the chip implemented this functionality in hardware.

Figure 4-2 and the activities involved. For example, we expect the instruction cache penalties from *Block Ops* to be very low, as this code is written as a tight loop and so has good instruction locality. In contrast, the data cache penalties from *Block Ops* are high. This is not a surprise; the bad data locality of these routines is well documented [6, 63, 78].

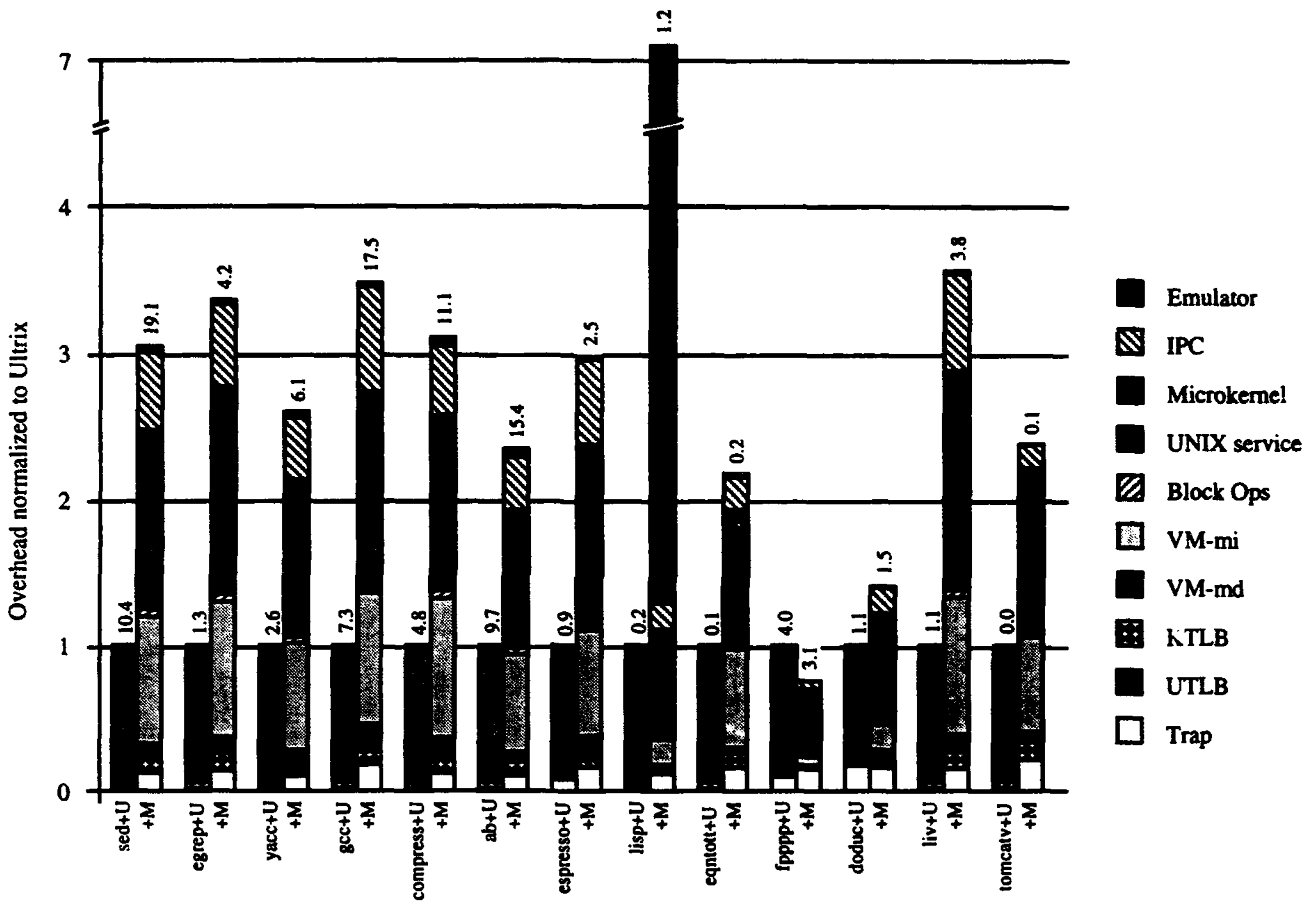

Figure 4-3: System instruction cache misses for Ultrix and Mach.

This figure shows the system instruction-cache overheads for programs running on Ultrix and Mach 3.0. Ultrix penalties are normalized to one. The number at the top of each bar is the percentage of total cycles that are due to the system instruction-cache misses.

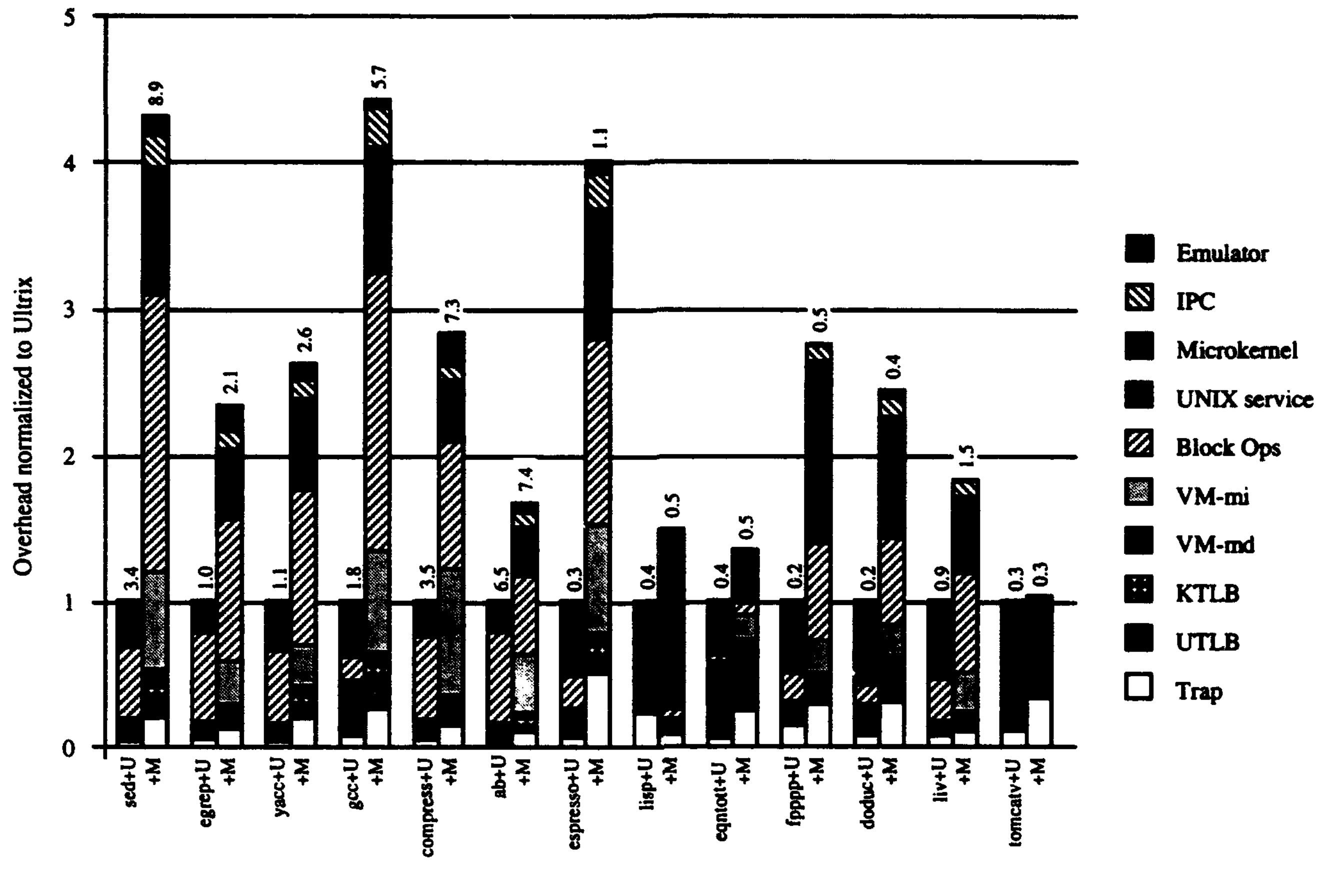

Figure 4-4: System data cache misses for Ultrix and Mach.

This figure shows the system data-cache overheads for programs running on Ultrix and Mach 3.0. Ultrix penalties are normalized to one. The number at the top of each bar is the percentage of total cycles that are due to the system data-cache misses.

4.3. Memory System Behavior: Seven Assertions

In this section we evaluate Ultrix and Mach 3.0 memory system behavior in terms of the seven assertions from Table 1-1. Our basic strategy is to consider each assertion in the context of the behavior measured in simulation experiments. In several cases we have used additional simulations, with variations in the base memory system that reveal the sensitivity of system performance to the assertion in question.

Assertion	Implication
1. The operating system has less instruction and data locality than user programs [24, 26].	The operating system isn't getting faster as fast as user programs.
2. System execution is more dependent on instruction cache behavior than is user execution [78].	A balanced cache system for user programs may not be balanced for the system.
3. Collisions between user and system references lead to significant performance degradation in the memory system (cache and TLB) [60, 78, 82].	A split user/system cache could improve performance.
4. Self-interference is a problem in system instruction reference streams [57, 78].	Increased cache associativity and/or the use of text placement tools could improve performance.
5. System block memory operations are responsible for a large percentage of memory system reference costs [63, 78].	Programs that incur many block memory operations will run more slowly than expected.
6. Write buffers are less effective for system (as opposed to user) reference streams [6, 34].	A write buffer adequate for user code may not be adequate for system code.
7. Virtual page mapping strategies can have significant impact on cache performance [47, 59].	Systems should support a flexible page mapping interface, and should avoid default strategies that are prone to pathological behavior.

Table 1-1: Seven assertions about the memory behavior of operating systems, repeated from page 3.

4.3.1. System and user locality

As cache behavior is an indication of locality, Table 4-1 supports the first assertion: *The operating system has less instruction and data locality than user programs.* The system can contribute up to 51% of non-idle instruction cache references, but in most cases (17 of 26) the system contribution is less than 10%. Given this observation, a disproportionately large number of instruction cache misses are due to the system (greater than 70% for two-thirds of the workload/system pairs).

In terms of data references, the system contributes a larger percentage of misses than references, again supporting the assertion that system data locality is worse than that of user activity. Even so, in only five of the workload/system combinations does the system contribute more than 90% of data misses, and only twelve if the threshold is lowered to 50%. Although the system's contribution of instruction and data references are comparable, the percentage of misses is not. Instruction references miss more often than data references for both Mach and Ultrix. From this we conclude that instruction locality is worse than data locality during system execution.

The percentage of instruction and data misses due to the system is generally larger under Mach than Ultrix. Figure 4-1 and Table 4-2 together show that the difference in user cache behavior between Ultrix and Mach is small. As Mach incurs a larger number of cache misses than Ultrix, and as nearly every additional cache miss is due to the system, the percentage of misses due to the system is larger.

4.3.2. System instruction locality

Percentages are useful for comparing system and user behavior but they cloud overall performance effects. For example, although 97% of instruction cache misses for *eqntott* under Mach are due to the system, the system instruction cache miss rate is insignificant.[10]

A better indicator of the performance impact of locality is the cache's contribution to *MCPI*. In Table 4-3 we combine our baseline data from Table 4-1 with cache miss penalties for the simulated memory system to yield the *MCPI* contributions from the cache. The component of *MCPI* due to system instruction cache references dominates that due to the user in 20 of 26 cases. In contrast, the system data cache component dominates in only thirteen cases. Furthermore, the majority of system (as opposed to user) cache penalties are due to poor instruction cache behavior; only five runs show greater penalties for data than instructions, and in these runs the system contribution to *MCPI* is small. This behavior supports the second assertion: *System execution is more dependent on instruction cache behavior than is user execution.* However, many of the programs in our workload have small working sets that fit entirely in the instruction cache. Larger programs which do not fit well in the cache, such as *gcc* and the realistic X11 workloads discussed in Chapter 6, have instruction cache penalties comparable to that of the system.

Table 4-3 quantifies the difference in *MCPI* between Mach and Ultrix that was represented visually in Figure 4-1. Memory penalties due to system instruction and sys-

[10]The system instruction cache miss rates can be calculated with data from Table 4-1 as the *number of system instruction cache misses / number of system instruction cache references*. For example, for *eqntott*:

$$\frac{254 \times 0.97}{1417868 \times 0.01} = 0.017$$

Similarly, the user instruction cache miss rate is nearly zero (0.0005%).

	instruction cache				data cache			
	Ultrix		Mach		Ultrix		Mach	
workload	sys	user	sys	user	sys	user	sys	user
sed	**0.129**	**0.005**	**0.283**	**0.005**	**0.041**	**0.001**	**0.132**	**0.003**
egrep	**0.014**	**0.001**	**0.046**	**0.001**	**0.010**	**0.000**	**0.023**	**0.000**
yacc	**0.028**	**0.004**	**0.069**	**0.003**	**0.011**	**0.011**	**0.029**	**0.012**
gcc	0.103	0.145	**0.294**	**0.123**	0.027	0.034	**0.094**	**0.039**
compress	**0.060**	**0.002**	**0.157**	**0.005**	0.042	0.106	0.101	0.102
ab	**0.139**	**0.130**	**0.261**	**0.098**	**0.091**	**0.024**	**0.121**	**0.020**
espresso	0.009	0.012	**0.026**	**0.011**	0.003	0.007	**0.011**	**0.008**
lisp	**0.002**	**0.001**	**0.013**	**0.011**	0.003	0.004	**0.006**	**0.003**
eqntott	**0.001**	**0.000**	**0.003**	**0.000**	0.005	0.147	0.006	0.147
fpppp	0.050	0.184	0.040	0.173	0.002	0.005	0.005	0.005
doduc	0.014	0.277	0.020	0.270	0.002	0.023	0.006	0.022
liv	**0.013**	**0.000**	**0.045**	**0.000**	**0.010**	**0.001**	**0.018**	**0.000**
tomcatv	**0.000**	**0.000**	**0.002**	**0.000**	0.005	0.634	0.005	0.634

Table 4-3: MCPI contributions from the cache.

For each workload/system pair, this table shows the *MCPI* component due to the instruction and data caches. Runs for which the system contribution to *MCPI* dominates that of the user are shown in boldface.

tem data references are larger for Mach than for Ultrix, while user memory penalties are similar. Increased system activity in Mach, as is shown in Figure 4-2, results in a larger cache contribution to *MCPI.*

4.3.3. Competition between user and system activity

The increased cache activity for Mach 3.0 suggests that workloads may have worse memory system behavior for Mach 3.0 than for Ultrix due to increased competition in the cache between active system and user contexts. To evaluate this, we modeled a hypothetical memory system where this competition was eliminated. We eliminated competition by duplicating the memory system, with one instance used for user activity and one for system activity. Apart from the elimination of competition the private memory systems were identical to that of the base simulation.

The effects of user/system competition on cache behavior are shown in Figure 4-5. Instruction and data cache behavior are each shown separately. Pairs of bars corresponding to Ultrix (+U) and Mach (+M) are aligned to emphasize the comparison between the two operating system implementations. Each instruction and data bar has four components. The two leftmost components correspond to the case of separate system and user caches, and represent the fraction of misses that remain with the independent memory systems. The two rightmost components show the additional fraction of system and user misses that occur when the cache is unified.

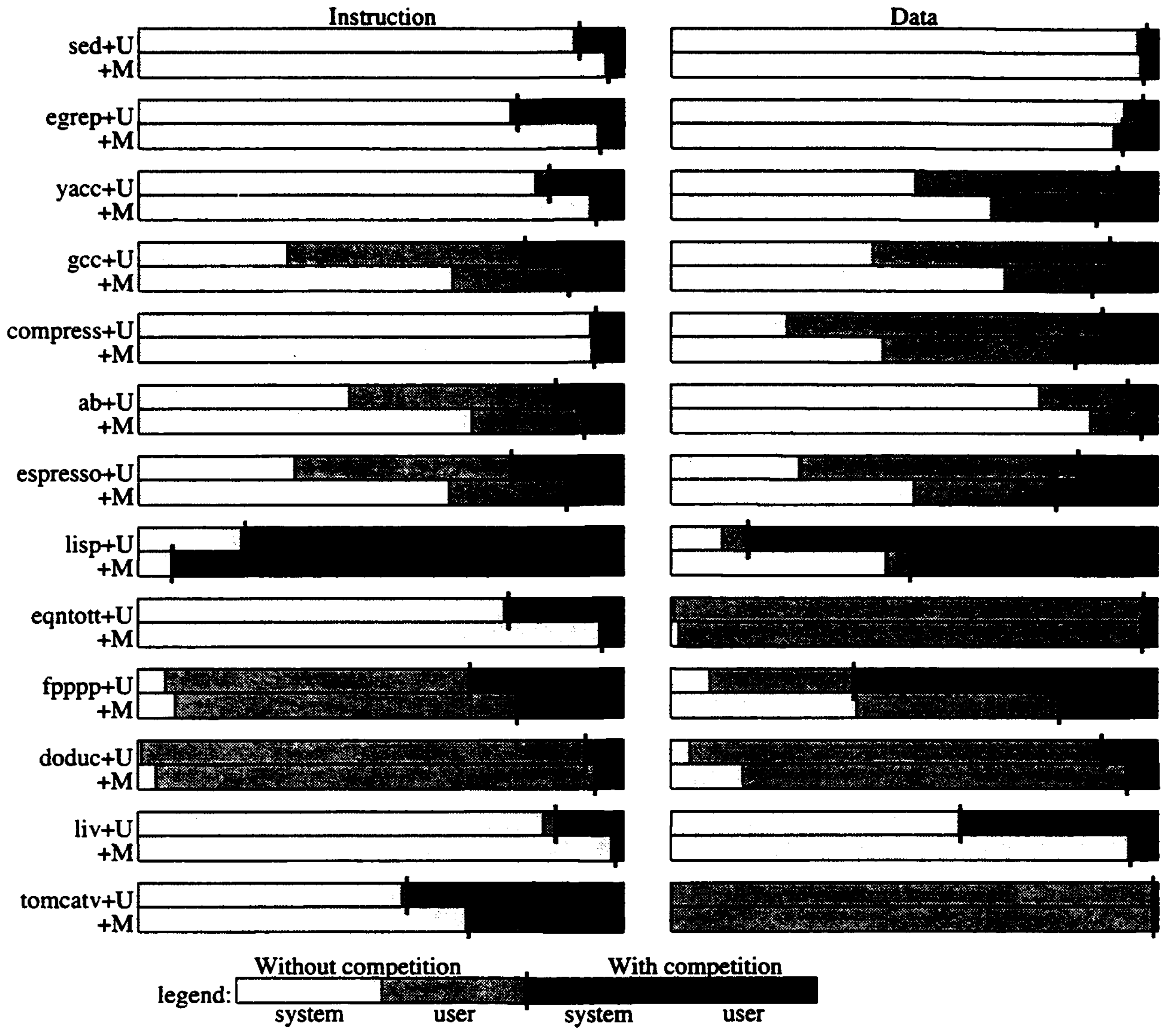

Figure 4-5: User/system interference.

For each workload/system pair, this figure shows effects of competition between system and user activity under Ultrix (+U) and Mach (+M). Instruction and data activity are shown separately. Each bar is composed of four regions. The two rightmost regions represent the fraction of misses that are due to competition. The two leftmost regions represent the fraction of misses that remain when user/system competition is eliminated.

Although our separate user and system caches double the effective cache size, the general dominance of the two leftmost components in Figure 4-5 indicates that the isolated caches do not significantly reduce miss rates relative to the unified cache. The largest interference effects (for example, those that occur for *lisp*) occur when the cache miss rate is low, such that a few interference misses can result in a large relative change.

The absolute contribution of competition misses to *MCPI* is shown in Table 4-4. These points imply that the third assertion: ***Collisions between user and system references lead to significant performance degradation in the memory system***, is not true for these workloads with the simulated memory system.

	Ultrix			Mach		
workload	inst	data	total	inst	data	total
sed	0.010	-0.006	0.004	0.009	0.004	0.013
egrep	0.003	0.000	0.003	0.002	0.002	0.004
yacc	0.005	0.002	0.007	0.004	0.005	0.009
gcc	0.050	0.007	0.057	0.047	0.018	0.065
compress	0.004	0.018	0.022	0.010	0.034	0.044
ab	0.038	0.006	0.044	0.029	0.000	0.029
espresso	0.005	0.002	0.007	0.004	0.004	0.008
lisp	0.002	0.006	0.008	0.022	0.004	0.026
eqntott	0.000	0.004	0.005	0.000	0.005	0.005
fpppp	0.072	0.002	0.074	0.047	0.002	0.049
doduc	0.023	0.002	0.025	0.016	0.002	0.018
liv	0.001	0.004	0.005	0.000	0.001	0.002
tomcatv	0.000	0.005	0.006	0.000	0.005	0.006

Table 4-4: MCPI contributions from cache competition.

This table shows *MCPI* contributions from additional system misses occurring when cache competition with user references is present. The negative value for *sed* running on Ultrix is because user references can actually reduce the number of system misses due to data that is shared between the user and system.

For user/system interaction in a Unix system, a voluntary context switch occurs for every system call. Table 4-4 shows that, for the workloads we consider, the additional competition cache misses following a voluntary context switch do not have significant impact on overall performance.[11] On the user side, where the instruction cache miss rates are generally low but data cache miss rates are high, the cost of reloading the cache after a context switch is amortized over a large number of instructions. On the system side, instruction and data locality are already poor, limiting the impact of interleaved user references. This behavior is consistent with earlier results on competition in client-server systems [57]. However, the penalty from competition clearly depends on the client-server system in question. In Chapter 6 we show that competition between kernel, server, and client contexts can have a significant impact on performance for X11 client/server workloads.

[11]We distinguish between competition from voluntary context switches, as occurs in a client-server system, and competition from involuntary context switches, as occurs in a multitasking workload.

TLB behavior

The Ultrix kernel binary runs in unmapped kernel memory, largely isolating it from the TLB. In contrast, only the Mach microkernel component runs unmapped; the UNIX server and emulator run in mapped memory. This induces user/system competition for TLB resources. Earlier research has shown that this competition can cause a significant increase in TLB activity [6, 60]. Table 4-5 confirms this, showing an order of magnitude increase in the number of system TLB misses for Mach as compared to Ultrix.

In terms of *MCPI*, though, the absolute contribution of system TLB misses to performance is generally not large, as shown by the last four columns of Table 4-5. Moreover, high TLB *MCPI* is an indication of poor locality, which is also reflected in more severe cache penalties. Even in runs with the most extreme behavior, TLB penalties are consistently dominated by cache penalties (Table 4-3) for both Ultrix and Mach.

	TLB refs (x1000)			UTLB misses (x1000)				KTLB misses (x1)		UTLB MCPI		KTLB MCPI	
workload	user	Ultrix	Mach	U-user	M-user	Ultrix	Mach	Ultrix	Mach	Ultrix	Mach	Ultrix	Mach
sed	5596	423	1079	0.08	0.49	0.05	6.67	427	2132	0.000	0.012	0.021	0.063
egrep	50399	546	1116	0.06	0.39	0.10	6.41	447	1847	0.000	0.002	0.003	0.009
yacc	37460	571	1323	0.22	1.26	0.05	7.89	373	2280	0.000	0.003	0.003	0.015
gcc	30093	1582	2951	28.78	35.87	0.10	17.87	1491	3305	0.011	0.018	0.016	0.022
compress	17892	986	2085	78.92	82.06	0.13	10.24	690	3982	0.045	0.045	0.011	0.045
ab	755092	90958	195492	1072.61	1208.04	12.50	1457.98	94635	578598	0.013	0.025	0.030	0.108
espresso	164313	660	1281	0.74	2.64	0.05	7.67	441	3111	0.000	0.001	0.001	0.005
lisp	1706833	12974	26783	0.07	12.69	0.04	15.68	392	8063	0.000	0.000	0.000	0.002
eqntott	1690678	3579	3697	675.05	692.57	0.11	24.03	1317	9760	0.005	0.007	0.000	0.002
fpppp	380307	3632	1169	3.00	13.54	0.25	9.02	361	2273	0.000	0.001	0.000	0.003
doduc	438563	899	2162	6.54	30.53	0.04	18.26	391	5811	0.000	0.002	0.000	0.005
liv	30123	232	417	0.03	0.11	0.04	2.62	184	701	0.000	0.002	0.002	0.007
tomcatv	2949614	4480	2684	317.74	321.79	0.13	25.69	1557	8135	0.002	0.003	0.000	0.001

Table 4-5: TLB activity.

This table shows TLB references (× 1000), UTLB misses (× 1000), KTLB misses (× 1), UTLB *MCPI*, and KTLB *MCPI* for system and user across the various workloads. The number of user UTLB references is the same for both systems, as the same user code is executed. UTLB miss counts depend on competition from the system, so the table shows separate numbers for Ultrix and Mach. KTLB misses do not occur in user code.

4.3.4. System self-interference

Self-interference occurs when insufficient cache associativity results in cache misses. The impact of self-interference in user-code is well-understood [43]. To evaluate the impact of system self-interference, we simulated a two-way LRU set associative cache of the same size as our direct-mapped cache. As in the previous section, user references are isolated from the system-only cache, although they continue to generate TLB misses and subsequent system activity.

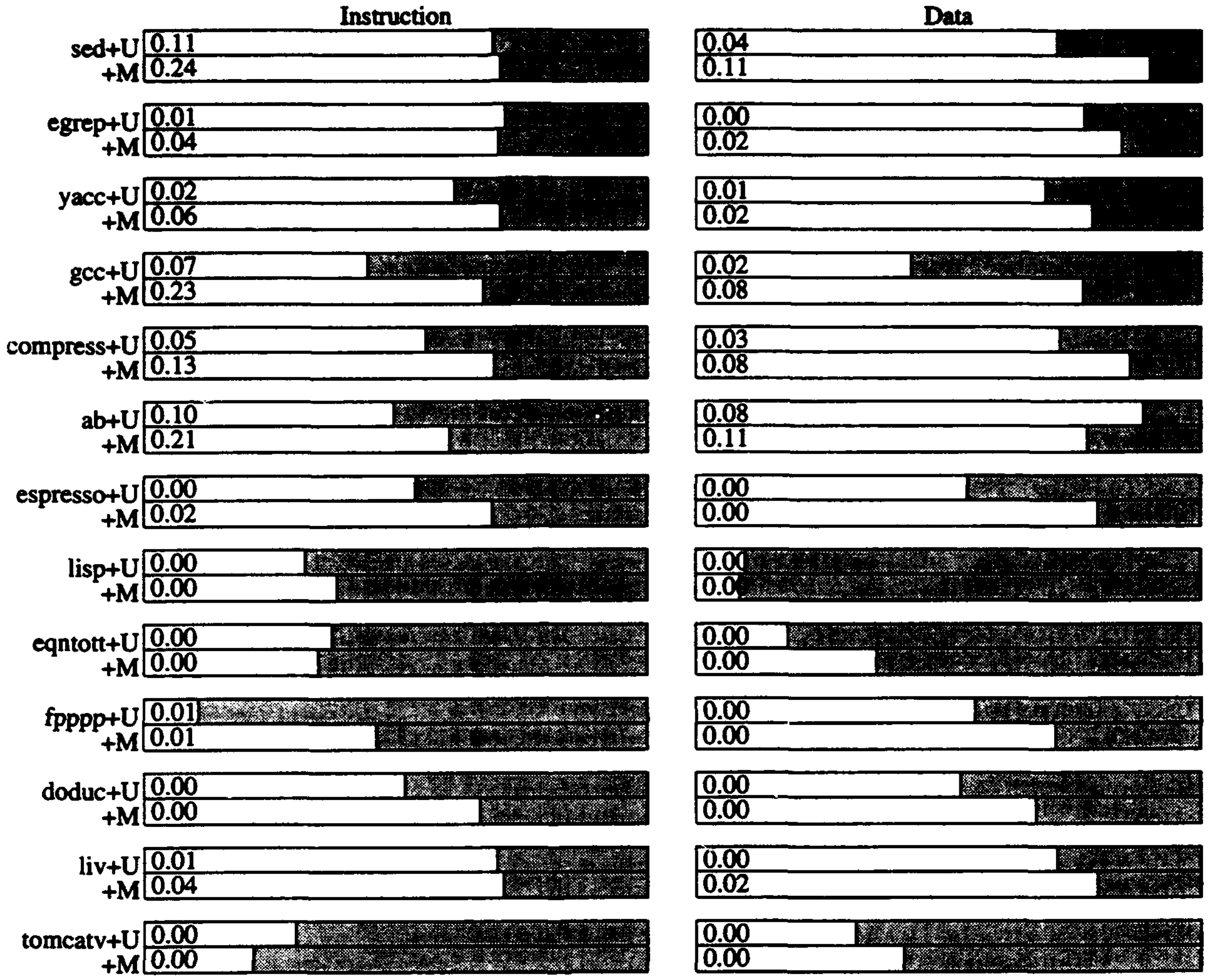

Figure 4-6: System self-interference.

For each workload/system pair this figure shows system self-interference effects, as indicated by miss rates from direct-mapped and two-way associative caches of the same size. Each bar is composed of two regions. The darker region represents misses eliminated by associativity (those due to self-interference). The lighter region represents misses that associativity does not eliminate. The number on the left end of the bar is *MCPI* for the system-only direct-mapped cache.

Figure 4-6 illustrates the effect of the increased system associativity on instruction and data cache miss rates. In each bar, the light region represents the fraction of system misses that associativity does not eliminate, while the dark region represents that fraction eliminated by associativity. This representation emphasizes variations in the relative benefit of associativity between workloads. The number at the left side of each bar is the absolute *MCPI* contribution of cache misses for a system-only direct-mapped cache. Figure 4-6 shows that the increased associativity eliminates a significant fraction of misses, and is more effective for instruction than data references. This confirms the fourth assertion: *Self-interference is a problem in system instruction reference streams*.

Self-interference has the largest relative impact when *MCPI* is low, and the smallest relative impact when *MCPI* is high. Associativity helps to eliminate collision misses but is of no benefit for capacity misses. A high *MCPI* occurs when the cache capacity is inadequate, a situation where increased associativity does not help. Examples are *sed*, *egrep*, and *liv*, which have high *MCPI*s, a large amount of system activity, and gain relatively little from associativity. In contrast, associativity helps most with *lisp* and *tomcatv*, where system activity is limited. Associativity is generally less beneficial for Mach than for Ultrix because the working set of system code and data tends to be larger for Mach.

4.3.5. Block operations

Operating systems perform block memory operations to transfer data between I/O devices and memory, and to copy data between address spaces. Table 4-6 shows that block memory operations and their subsequent interference can be responsible for a substantial fraction of total *MCPI*, especially for programs that perform significant I/O. *Espresso*, while not I/O intensive, pays a high relative penalty for block operations because program loading overheads dominate its cache behavior. From the measurements we conclude that assertion five: *System block memory operations are responsible for a large percentage of memory system reference costs* is true, and most important in I/O intensive applications.

In terms of *MCPI*, Table 4-6 shows that block operations incur a larger absolute overhead for programs running on Mach than on Ultrix. Table 4-7 shows that Mach generally references more data than Ultrix in block operations and that more of those references go through to memory. Block operations in Mach occur within the kernel as part of the VM and IPC systems, and within the UNIX server as part of the file system. In contrast, Ultrix block operations, which occur entirely within the kernel, are due mostly to VM and file system operations.

	Ultrix		Mach	
workload	MCPI	%total	MCPI	%total
sed	0.066	29.2	0.131	26.6
egrep	0.014	39.3	0.017	20.9
yacc	0.017	25.6	0.027	20.9
gcc	0.116	26.8	0.159	23.0
compress	0.055	22.1	0.071	17.0
ab	0.100	23.4	0.057	10.7
espresso	0.009	21.3	0.013	19.9
lisp	0.000	0.3	0.000	0.0
eqntott	0.000	0.4	0.001	0.6
fpppp	0.003	1.2	0.005	2.2
doduc	0.003	0.9	0.006	1.9
liv	0.008	7.1	0.013	7.9
tomcatv	0.000	0.0	0.000	0.0

Table 4-6: MCPI from block memory operations.

For each system, this table shows the *MCPI* contribution of block moves (and subsequent interference), and also the percentage of total *MCPI* due to block moves.

	Ultrix						Mach					
	MCPI		data reads		memory reads		MCPI		data reads		memory reads	
workload	B-Ops	%total	cacheable	uncacheable	total	%	B-Ops	%total	cacheable	uncacheable	total	%
sed	0.066	29.2	57	17	28	37.7	0.131	26.6	132	26	70	44.2
egrep	0.014	39.3	88	19	42	40.0	0.017	20.9	126	15	51	36.6
yacc	0.017	25.6	105	27	42	32.3	0.027	20.9	136	27	64	39.6
gcc	0.116	26.8	53	253	277	90.3	0.159	23.0	237	289	414	78.6
compress	0.055	22.1	168	35	68	33.5	0.071	17.0	180	45	98	43.5
ab	0.100	23.4	16729	1897	6118	32.9	0.057	10.7	10311	609	4442	40.7
espresso	0.009	21.3	43	80	93	75.7	0.013	19.9	143	87	133	57.8
lisp	0.000	0.3	1	2	3	100.0	0.000	0.0	76	0	0	1.0
eqntott	0.000	0.4	258	23	56	20.0	0.001	0.6	232	0	94	40.4
fpppp	0.003	1.2	19	63	71	84.8	0.005	2.2	125	69	99	51.1
doduc	0.003	0.9	36	62	75	76.5	0.006	1.9	115	85	147	73.2
liv	0.008	7.1	19	8	14	51.1	0.013	7.9	52	8	20	34.2
tomcatv	0.000	0.0	113	23	60	44.5	0.000	0.0	297	4	76	25.3

Table 4-7: Block memory operations and memory reads.

For each system, this table shows the *MCPI* due to block memory operations and subsequent interference, and its percentage of total *MCPI* (Figure 4-1). The table also shows the number of data reads from cacheable and uncacheable memory that are due to block operations, the number of those reads that go to memory resulting in a CPU read stall, and the percentage of overall CPU memory stalls due to block operations. Reads from uncacheable memory are due primarily to I/O operations and always go through to memory. All counts are in thousands.

4.3.6. Streaming writes

Operating systems stream data to memory during block transfers, such as for I/O and IPC, and during context switches and exception handling. Write buffers expedite streaming writes by retiring writes more quickly and by allowing the CPU to run ahead of memory. The effect of streaming write operations on system performance can be measured by counting stall cycles due to writes. The number of write stall cycles per instruction for user and system code under Ultrix and Mach is shown in Table 4-8. In most cases system behavior is worse than user behavior, supporting the sixth assertion: *Write buffers are less effective for system references.*

	Ultrix		Mach	
workload	system	user	system	user
sed	**0.061**	**0.000**	**0.076**	**0.000**
egrep	**0.050**	**0.002**	**0.065**	**0.002**
yacc	**0.062**	**0.000**	**0.076**	**0.000**
gcc	**0.106**	**0.012**	**0.129**	**0.012**
compress	**0.043**	**0.011**	**0.063**	**0.013**
ab	**0.040**	**0.009**	**0.043**	**0.010**
espresso	**0.093**	**0.001**	**0.111**	**0.001**
lisp	**0.007**	**0.004**	**0.064**	**0.005**
eqntott	**0.014**	**0.000**	**0.024**	**0.000**
fpppp	**0.030**	**0.017**	**0.037**	**0.015**
doduc	**0.101**	**0.018**	**0.095**	**0.018**
liv	0.052	0.090	0.075	0.090
tomcatv	0.023	0.033	**0.044**	**0.033**

Table 4-8: Write buffer stall cycles per instruction.

This table shows write buffer stall cycles per user instruction and write buffer stall cycles per system instruction. Runs in which system behavior is worse than user behavior are shown in bold face.

User write buffer stalls per instruction are generally higher for Mach than for Ultrix. This might seem anomalous, as both systems use the same user application with the same input. The difference is due to interactions between the cache and the write buffer. Overall cache miss rates are higher with Mach, and the DECstation 5000/200 memory system gives CPU reads priority over outstanding writes. Consequently, fewer memory cycles are available for the write buffer to retire outstanding writes, resulting in a larger number of stalls. Additionally, the interleaved read misses decrease the frequency of low-latency page-mode writes. .

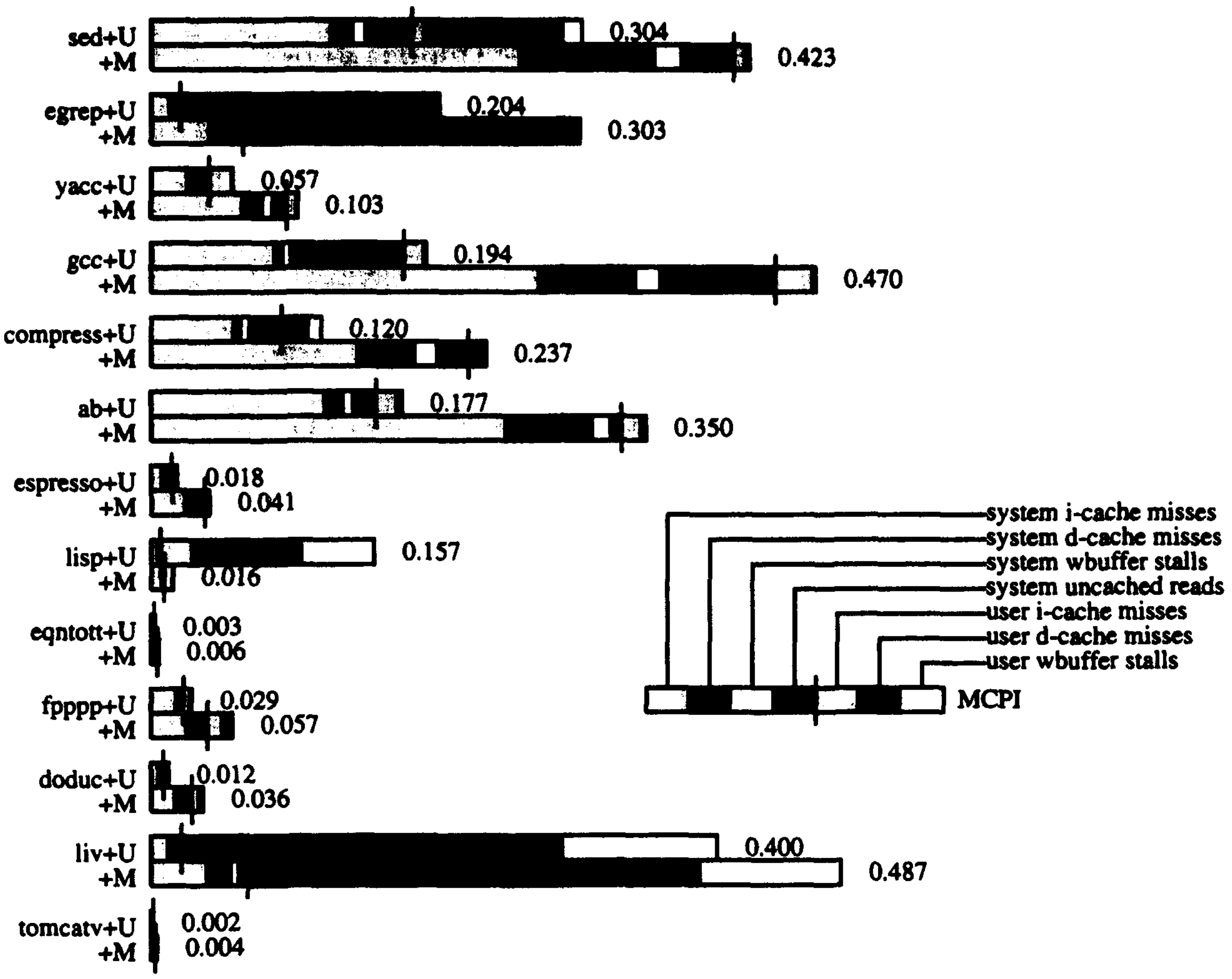

Figure 4-7: MCPI for random page mapping.

This figure shows *MCPI* for Mach and Ultrix, as in Figure 4-1, but for a system that uses random page mapping. The elimination of user cache misses reduces memory contention, so write buffer stalls are virtually eliminated for many workloads. This figure shows the results of a single run. As the page-mappings are random, behavior can vary significantly between runs.

4.3.7. Page mapping policy

The system's virtual page mapping policy can affect the performance of a physical cache because it determines the placement and overlap of virtual pages in the cache. As an example, consider a deterministic policy which selects mappings such that pages contiguous in the virtual address space are also contiguous in the physical cache. Such a policy can eliminate self-conflict misses for applications that are smaller than the cache. It also improves the effectiveness of program reordering tools that rearrange the layout of text and data in memory to improve cache performance [32, 55].

In our simulations to compare the memory behavior induced by Ultrix and Mach 3.0 system structure, we have used the same deterministic page-mapping policy for both the Ultrix and Mach experiments. As mentioned earlier, Ultrix uses a deterministic strategy while the Mach 3.0 strategy is essentially random (a virtual page is assigned to the next physical page on the free list). To isolate the effect of the page mapping strategy, we modified our simulator to use a random mapping policy. The simulator maintains a page table so that when a mapping is created for a virtual page it does not change. Figure 4-7 shows *MCPI* for a run of the workloads with random page mapping. When compared to Figure 4-1 (both are on the same scale), most of the workload/system pairs perform better with random page mapping, and *gcc, compress, eqntott, fpppp, doduc* and *tomcatv* show the greatest improvement. The program *tomcatv* offers a good example of the effect that mapping strategy can have on program performance. This program uses several matrices that are rough multiples of the cache size, and are allocated contiguously in virtual memory. The virtual-to-physical mapping induced by the deterministic strategy causes frequent collisions between corresponding matrix elements during computation.

In some cases the deterministic strategy yields a page mapping with low user cache miss rates. Specific examples are *sed* and *lisp* under Ultrix, and *egrep* and *liv* for both systems. Note that with the small instruction text of these programs, the mapping chosen by the deterministic policy is optimal for the instruction cache. In these cases the deterministic strategy leads to good overall behavior, while the random strategy can perform significantly worse. Our results suggest that such cases are infrequent for the memory system we simulated, and the deterministic policy frequently causes worse behavior than a random policy.

Overall, these observations confirm the seventh assertion: *Virtual to physical page mapping strategy can have significant impact on cache performance.* Moreover, a deterministic strategy can have a negative impact on performance for a direct-mapped cache when program reordering tools are not used. In such cases a random strategy is less likely to induce consistently poor behavior.

4.4. Conclusions

For the majority of workloads we consider, the number and cost of non-idle instructions executed is substantially higher for Mach than for Ultrix. Six of the assertions about operating systems and memory system behavior are true, although two have little or no impact on system performance. One is false. Several are sensitive to the operating system architecture. Specifically:

- **System and user locality.** System locality is measurably worse than user locality, and the performance impact can be significant. The Mach microkernel-based system has poorer system locality than Ultrix.
- **System instruction locality.** Relative to user behavior, system text shows less locality than system data. However, user workloads such as *gcc* with large text can have instruction cache penalties that rival that of the operating system.
- **User/system competition.** User/system competition is a measurable component of cache and TLB miss rates. For these workloads, though, system performance is not affected by user/system competition. The impact of the Mach 3.0 microkernel structure on competition is not significant.
- **System self-interference.** Self-interference accounts for a significant number of system misses, particularly in system text. However, the cases with the worst overall behavior are also those that benefit least from associativity. Compared to Ultrix, associativity eliminates a lower percentage of Mach's cache misses because of its greater demand for cache resources.
- **Block operations.** Block operations can be responsible for a large component of overall *MCPI*, particularly for applications that perform I/O. Mach moves more data with block operations and has a larger *MCPI* due to block operations than Ultrix.
- **Streaming writes.** System code presents a higher load to the write buffer than user code. Mach's increased cache *MCPI* results in a larger number of system and user write buffer stalls due to competition between memory reads and writes.
- **Page mapping strategy.** Page mapping strategies can have a large effect on cache performance. The page mapping strategy is independent of operating system structure.

The performance of the operating system, either monolithic or microkernel-based, is more sensitive to memory system latency than that of applications. The locality of system code and data is inherently poor, and changes to memory systems that help application performance by taking advantage of locality are unlikely to bring proportional improvements to the system.

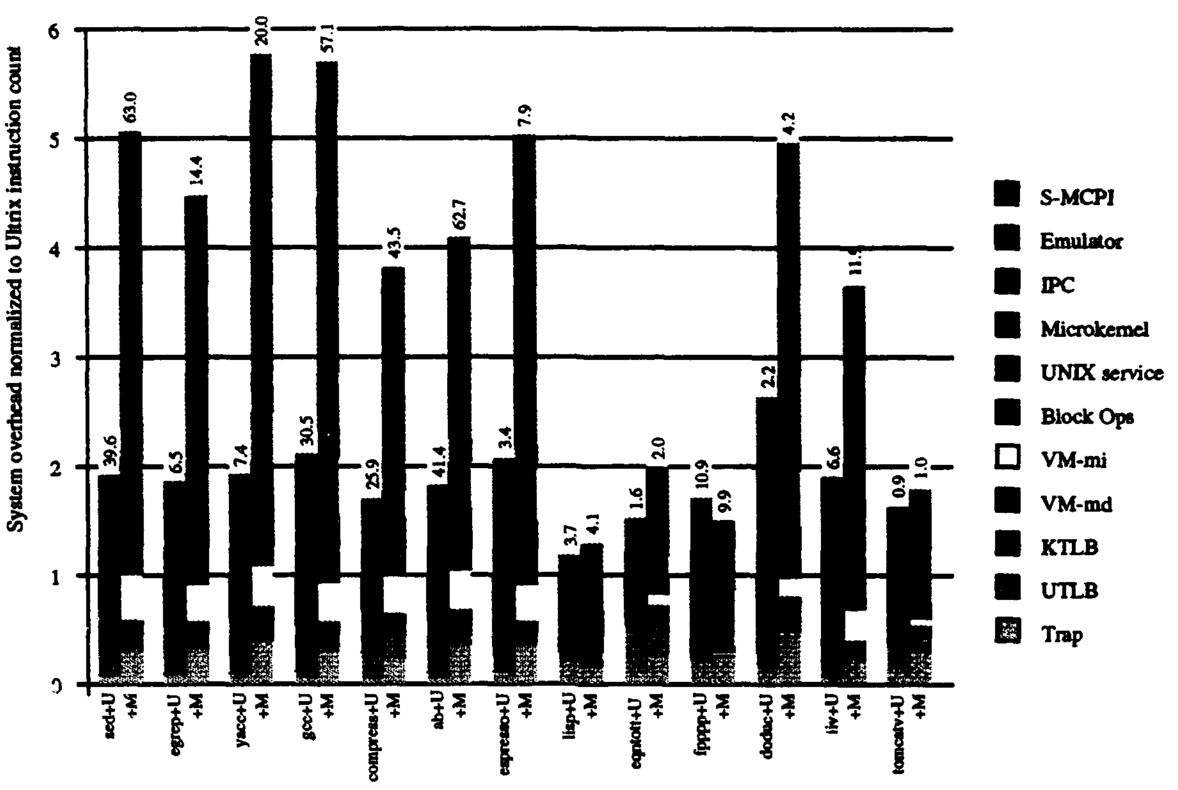

Figure 4-2. Relative System Overheads for programs running on Ultrix and Mach.

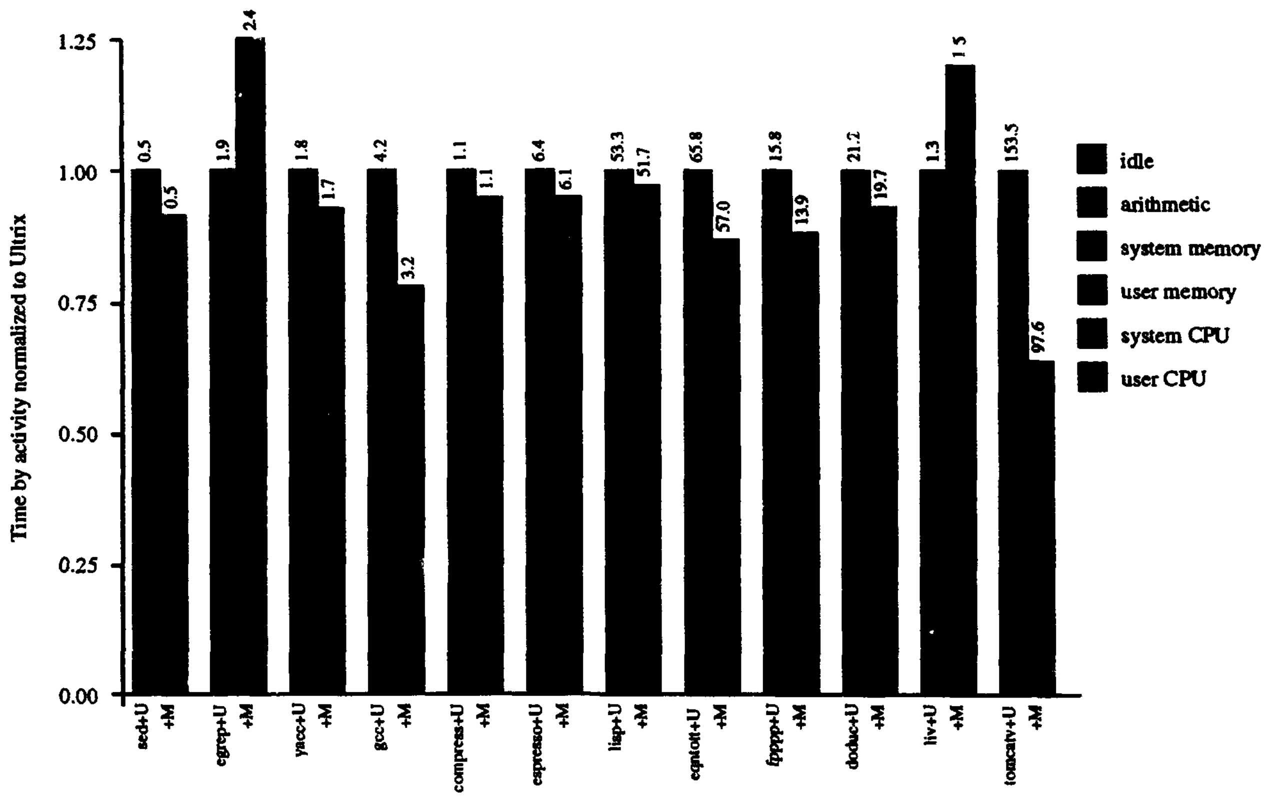

Figure 5-3. Ultrix and Mach 3.0 Execution Time; Page mapping policy as implemented.

Chapter 5
Mach 3.0 and Ultrix: The Impact of Policy

In this research we have categorized the principle differences between Mach 3.0 and Ultrix as either differences in *structure* or in *policy*. A difference is structural when it significantly impacts the sequence of instructions issued during (non-idle) operating system activity. Such differences affect system functionality, where functionality is implemented, or how different parts of the system interact at the software level. Structural differences have significant and broad-reaching impact on system implementation.

When a difference is not structural we say it is a difference in *policy*. Differences in policy cause no changes or only local changes in non-idle system activity. Ideally, policy is determined by code that is localized and isolated in the system, such that policy can be changed without changing other parts of the system.

In Chapter 4 we saw how structural differences between Ultrix and Mach 3.0 induce significantly higher instruction counts and memory penalties for system activity under Mach 3.0. In this chapter we place structural effects in the context of overall behavior and show the impact of system policy. We will explore two specific policy issues relevant in the comparison of Mach 3.0 and Ultrix: disk management policy and page mapping policy.

5.1. The Impact of Structure on Execution Time

Figure 5-1 shows simulator estimates of execution time for the experimental workloads[12] broken down into six classes of activity. For each workload there are two bars, one for Ultrix and one for Mach 3.0. For each workload, the bars are normalized to

[12]The Andrew Benchmark is not included. As we do not trace the assembler, simulator predictions of execution time are not possible.

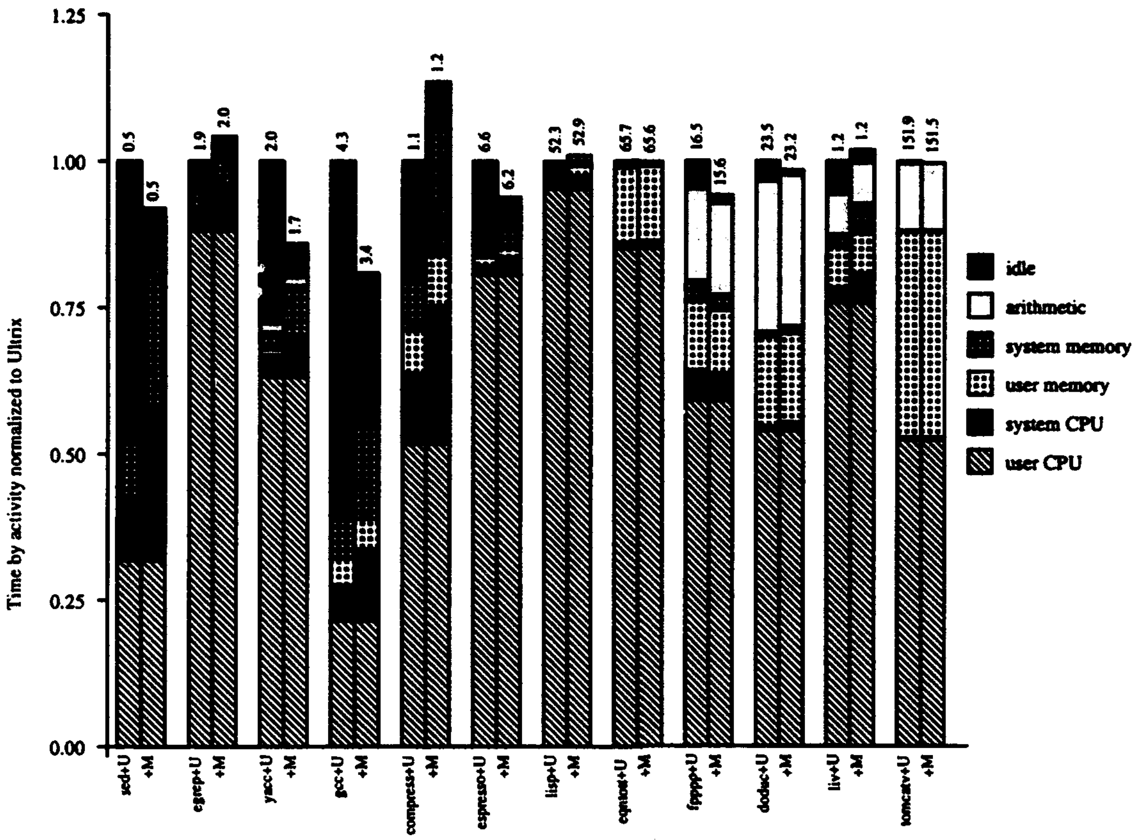

Figure 5-1: Ultrix and Mach 3.0 Execution Time: The Impact of Structure

This figure compares simulator estimates of execution time for Ultrix and Mach 3.0. Execution time is broken down into six sources of latency, corresponding to user and system CPU cycles, user and system memory cycles, arithmetic stalls, and idle time. Both Mach 3.0 and Ultrix simulations used the same deterministic page mapping algorithm. This figure illustrates the significant performance penalties induced by Mach 3.0 structure, both from increased system instruction counts and increased memory system penalties. It also shows the impact of the conservative Ultrix disk policy, with significantly more idle cycles for Ultrix than Mach 3.0.

execution time under Ultrix. The number at the top of each bar is elapsed time as predicted by the simulator.

Starting from the bottom of the graph and working up, the first two categories correspond to CPU cycles, that is, one cycle per instruction for each instruction executed by the processor. For an imaginary machine with no cache misses, a zero latency disk, and no arithmetic stalls, CPU cycles would be the only component of execution time. The first class of activity is *user CPU* cycles, that is, the one cycle for each user instruction executed on the CPU. Next is *system CPU* cycles, corresponding to system instructions executed. For a given workload, user CPU cycles are the same for Ultrix and Mach 3.0. User instruction counts are the same for both systems because each used the same user binary with the same input. For system intensive workloads such as *sed* and *egrep*, system CPU cycles for Mach 3.0 are generally about twice that of Ultrix. This means that Mach 3.0 executes about twice as many system instructions as Ultrix, an indication of the impact of system structure. Note that Mach 3.0 requires less system instructions for *lisp* and *fpppp*. See Section 4.2.1 for more details on this behavior.

The next two categories correspond to memory delays, including cache misses, write buffer stalls, and (for the system) uncached memory reads. In general, *user memory* delays are comparable for Ultrix and Mach 3.0. However, *system memory* delays are much higher for Mach 3.0, particularly for system-intensive workloads such as *sed*, *gcc*, and *compress*. This reflects the combined effect of poorer system locality and higher instruction counts for Mach 3.0, and again is an indication of the impact of system structure. For Figure 5-1, the same deterministic virtual-to-physical page mapping policy was simulated for both systems. Later in this chapter we will see that the different page mapping policies implemented in the two systems have a significant impact on cache behavior.

The next category corresponds to *arithmetic* stalls. These occur in user-mode only, as the operating system makes no use of floating point and rare use of multiply and divide instructions. The contribution to execution time is significant for floating point intensive workloads such as *tomcatv* and *doduc*. For integer workloads such as *gcc*, arithmetic stalls are not a significant source of latency as arithmetic instructions are rare.

Overall, Figure 5-1 shows that the structural differences between Mach 3.0 and Ultrix have a significant impact on overall behavior for many of the experimental workloads, both in terms of system instructions executed and system-induced memory delays.

5.2. Disk Policy

The last source of latency in Figure 5-1 corresponds to *idle* time, time spent in the system idle loop. For the workloads we consider, all idle loop activity corresponds to time spent waiting for synchronous disk operations to complete. Disk policy for Ultrix is different from that of Mach 3.0, causing Ultrix to spend more time in the idle loop. (Differences between disk management policy in the two systems were described in Section 3.5.3.) We classify disk management as a policy issue rather than a structural difference because it has little impact on non-idle operating system activity. It is straightforward in either system to implement either the Ultrix or Mach 3.0 policy. For example, Ultrix could be modified to use the Mach 3.0 policy by changing less than ten lines of code.

Figure 5-1 shows that synchronous disk activity, as reflected by time spent in the *idle* loop waiting for synchronous disk requests to complete, represents a significant component of execution time for workloads such as *sed* and *gcc*. Further, disk policy has a significant impact on this activity and on overall execution time.

5.3. Page Mapping Policy

Figure 5-1 reflects the impact of system structure and of disk policy on overall behavior. However, it does not reflect behavior and latencies in the real system, as can be seen by comparing times from Figure 5-1 and Table 3-2. The Mach 3.0 simulations in Figure 5-1 use a deterministic page mapping policy. This facilitates a comparison of the structural differences between Mach 3.0 and Ultrix in isolation from differences in policy. However, the page mapping policy implemented in Mach 3.0 is not deterministic. Mach 3.0 selects a physical page to back a given virtual page by taking the next page off the free list. The Mach 3.0 policy starts out as sequential, as pages on the free list are in order at boot time. As processes are created and destroyed and the free list becomes shuffled, the page mapping policy becomes random.

In Section 4.3.7 we saw that a random page mapping policy can sometimes give better memory system performance than the mapping induced by a deterministic policy as used in Ultrix. This occurs when the deterministic policy causes frequently referenced data to conflict in the cache. Figure 5-2 shows the effect of page mapping policy on execution times for Mach 3.0. For each workload, the bar on the left shows Mach 3.0 simulation results for a deterministic policy and the bar on the right shows the results for a random policy. The effect of page mapping policy on system cache behavior is relatively small, as many system memory references are unmapped and the proportion of capacity to compulsory misses is high. In contrast, many workloads have better user cache behavior for the program execution using a random policy than for the run with the

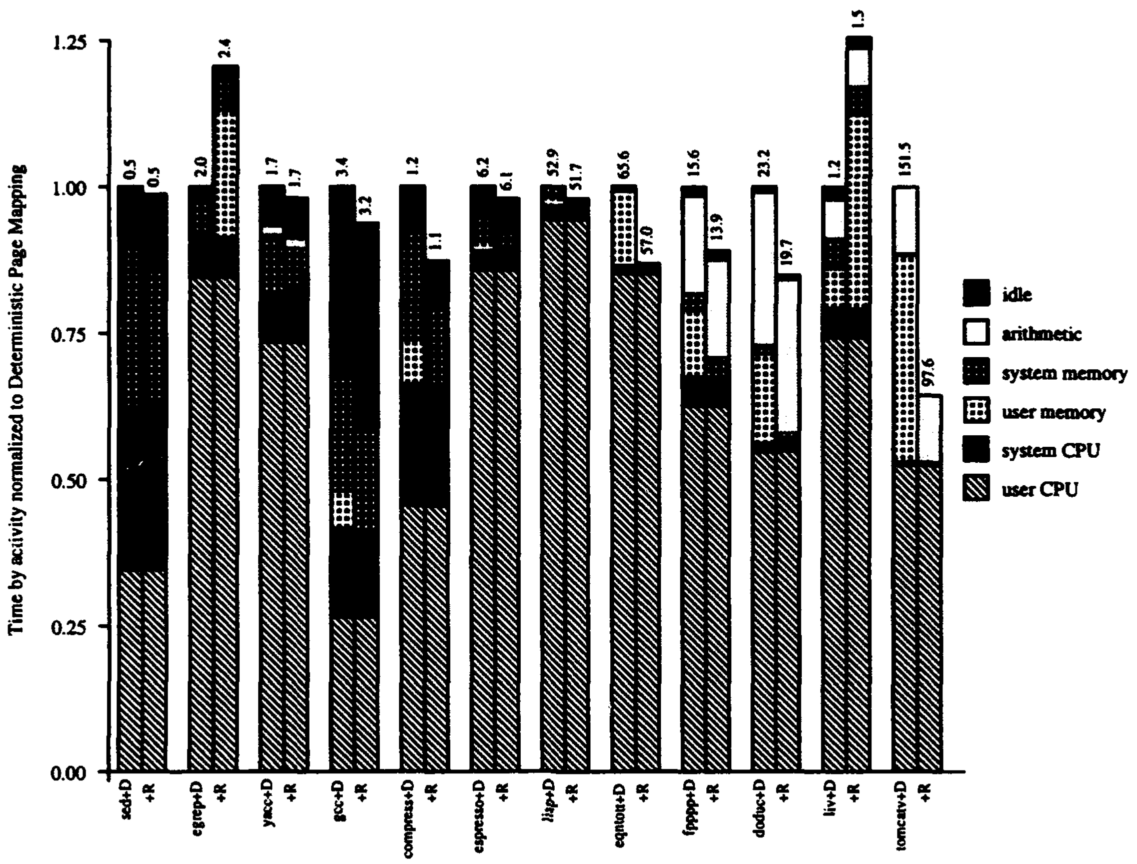

Figure 5-2: Page Mapping Policy and Mach 3.0 Execution times.

This figure shows the impact of page mapping policy in the context of Mach 3.0, comparing program execution with a deterministic page mapping policy to execution where page mappings are assigned by a random number generator. Execution time is broken down into six sources of latency. This figure illustrates the significant impact of page mapping policy on memory system delays. For many workloads, the performance of the deterministic policy is worse than that of a random page assignment policy, due to induced cache effects. Each bar shows activity during a single program execution.

deterministic policy. There are also cases where the random policy is worse, as in the experimental runs for *egrep* and *liv*.

For an intuition as to why the deterministic policy frequently gives worse cache behavior, it is useful to consider a combinatorial analysis of the page mapping policies. Suppose a program uses n pages that are consecutive in the virtual address space, and that each page can be mapped into one of k pages in the cache. Then there are k^n possible page mappings. With the random policy, each mapping has probability $1/k^n$ of being chosen. The k^n mappings can be partitioned into classes such that mappings in the same class cause the same pages to overlap in the cache, and hence have the same cache behavior. Because mappings in the same class have the same cache behavior, they can be said to be *equivalent*.

The deterministic policy determines a single class of equivalent mappings that will always be used. Note that with the deterministic policy, the placement of the first page determines the placement of all other pages. As there are k ways of placing the first page in the cache, the deterministic policy determines k possible mappings. Note that the random policy will choose a mapping equivalent to that of the deterministic policy with probability $\frac{k}{k^n} = \frac{1}{k^{n-1}}$.

Behavior with the random policy can vary widely across different runs, but the expected behavior corresponds to "average" behavior over the probability density function determined by all possible page mappings.

For certain workloads, the mapping determined by the deterministic policy gives worse-than-average behavior. In this case, the random policy will usually give better performance than the deterministic policy. The experimental results given in Figure 5-2 suggest that the deterministic policy frequently makes a worse-than-average choice.

Note from Figure 5-2 that neither the random nor the deterministic policy are consistently better across all the workloads. These policies are *static* in that the page mapping, once determined, cannot change in response to cache miss activity. Recent work in *dynamic* page mapping policies that update the virtual-to-physical page mapping in response to cache miss activity [14] shows promise in addressing this problem.

Figure 5-3 compares Ultrix and Mach 3.0 behavior using page mapping policies corresponding to those implemented in the actual systems. The Ultrix simulations use the deterministic page mapping policy and the Mach 3.0 simulations use a random policy. Figure 5-3 shows that the mapping has substantial impact on execution time. For many workloads the combined effects of page mapping and disk policy more than compensate for the performance penalties induced by system structure. As an example, system instruction overhead for the `compress` run on Mach 3.0 is twice that of the Ultrix run, but the Mach 3.0 run makes up for the difference through reductions in user cache cycles (page mapping policy) and idle time (disk policy). The breakdown in Figure 5-3 more accurately reflects the observed difference in execution time between Ultrix and Mach 3.0. The execution times in Figure 5-3 are still simulated, and are subject to variation and error from several sources. See Section 3.5.1 for details.

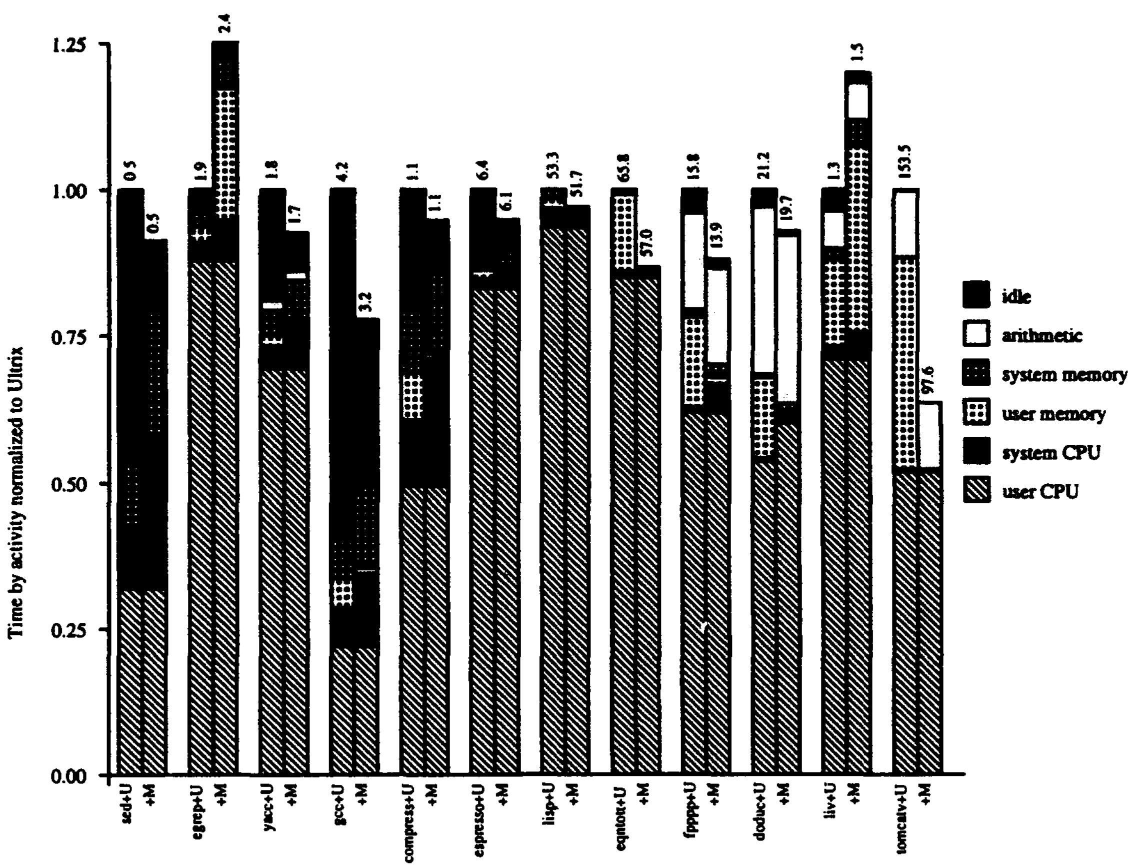

Figure 5-3: Ultrix and Mach 3.0 Execution Time; Page mapping policy as implemented.

See page 71 for a color version of this figure. This figure shows simulator estimates of execution time broken down into six sources of latency. The simulations used paged mapping policies as implemented in the system, deterministic for Ultrix and random for Mach 3.0. This figure shows how performance penalties induced by Mach 3.0 structure are countered in part by the effects of the Mach 3.0 disk and page mapping policy. Each bar shows activity during a single program execution.

As with disk policy, we have distinguished page mapping policy from *structural* differences between the two systems because of its minimal influence on the actual operating system implementation. Ultrix can be made to use the Mach 3.0 page mapping policy by modifying one line of source code.

5.4. Conclusions

In this chapter we have considered the impact of system structure and of system policy in the context of overall behavior, as reflected by execution time. We have seen that system structure has a large impact on overall behavior, both in terms of memory penalties and system instruction counts. As compared to Ultrix, the structural differences in Mach 3.0 induce substantial penalties. We completed the performance picture by considering the impact of system policy. Disk and page mapping policy for Mach 3.0 is different than that of Ultrix. These differences tend to improve Mach 3.0 performance and compensate for the penalties imposed by Mach 3.0 structure.

Chapter 6
X11 Workloads

The workloads used in the comparison of Mach 3.0 and Ultrix in the previous chapters are typical of those used for performance comparisons between different computer systems. In this chapter we look at workloads that use the X11 windowing system to understand how X11 workloads differ from those more traditionally used in performance evaluation. We used memory reference traces from DEC Ultrix, the X11 window system from MIT Project Athena, and freely available X11 applications to explore several aspects of memory system behavior and performance. The tracing system permits us to consider not only behavior within the X11 server but also interaction in the memory system between operating system, window server, and client.

Our analysis shows that memory behavior for X11 workloads differs substantially from that of traditional workloads, particularly in the instruction cache and TLB. Competition within and between contexts in the instruction cache has significant performance impact. This cache competition appears difficult to avoid in a direct mapped cache, suggesting that higher associativity may be required. TLB designs that do not accommodate the demands of large interactive systems may also become performance problems.

6.1. Background

X11 workloads, as compared to the SPECmarks [76] and other more traditional workloads for system performance comparison and analysis, differ in several fundamental ways:

- **Large program text.** Even the largest SPECmarks are small compared to X11. At 688K bytes, *gcc* stands out among the SPECmarks for its large text segment[13]. X11 servers commonly have as much as 1.8 megabytes of text, more than twice that of *gcc*. X11 clients also tend to have large code. The two real-world X11 clients used in this study, *gs* and *splot*, have text sizes of 946K bytes and 278K bytes respectively. User text size for *gs* with the X11 server is over four times that of *gcc*.

- **Three interacting contexts.** Typically, batch-oriented workloads involve two contexts: the user application and the kernel. For many of these workloads kernel activity is negligible. Scientific workloads are the most common examples. In contrast, activity in most X11 workloads is split among three contexts: the client, the X11 server, and the operating system, with significant activity occurring in all three contexts. The result is additional resource competition that does not happen in the two-context and single-context case.
- **Mandatory and potentially frequent context switches.** When multi-task workloads are used in memory system studies, they are usually created by taking unrelated batch-oriented workloads and running them simultaneously. Context switches for these multi-task workloads can often be scheduled arbitrarily. An intelligent scheduler may try to make switches infrequent as a strategy for minimizing cache competition. In contrast, scheduler policy is irrelevant in client-server systems. Context switches are largely determined by client behavior and inter-process communication implementations. Depending on the client, context switches may be frequent.

Additionally, the X11 server and clients are used daily and repeatedly by a large contingent of the workstation computing community. Many of these users make rare use of programs such as those in the SPECmarks.

Performance for benchmarks is typically measured in terms of *throughput*, with program execution times reduced to units such as MIPS or MFLOPS. A key distinction between interactive workloads and more traditional benchmarks is their sensitivity to *latency*, which is the time required for the system to respond to a given input event. Analysis of memory system components such as caches and write buffers is common practice for throughput benchmarks [22, 23, 38]. However, interactive programs and client-server systems have received relatively little attention in recent research [15, 58].

[13]Text sizes are given for Ultrix DECstation executables. *gcc* is as built from the SPECmark distribution. The X11 server size is for `/usr/bin/Xws` on an Ultrix workstation, which includes a number of DEC extensions. The tracing experiments used a smaller server (958K bytes). See Section 6.2 for details on the server used in the tracing experiments.

This is unfortunate in that, for many computer users, quick response time for latency-critical interactive applications is more important than the throughput of batch jobs. Because of the size and complexity of server-based systems such as X11, few detailed measurements of their behavior have been made. We think the problem deserves more attention, as memory system delays can have a significant impact on latency for interactive workloads.

6.1.1. Related Work in Measuring X11 Performance

This research focuses primarily on measuring the behavior and performance of realistic X11 client workloads from the perspective of the memory system. Several prior studies measured X11 behavior, although they differ substantially in that they considered behavior at higher levels of abstraction. Researchers at the Microelectronics Computation and Technology Corporation built a tool called XSCOPE to measure X11 performance and localize performance problems [65]. XSCOPE provides information about X11 request, reply, error, and event packets. Their experience in designing XSCOPE indicated some problems with the syntax of the X11 protocol.

Simple measures of performance, such as operations per second, are often used when characterizing new graphics hardware. Researchers at DEC WRL have done significant work in achieving good X11 performance, both with simple bit-mapped framebuffers [53] and more complicated hardware [54]. They also demonstrate software algorithms that permit effective use of the hardware. They consider memory reference behavior, but strictly as related to frame buffer references; application performance is beyond the scope their work. Researchers at Hewlett Packard used a technique called Direct Hardware Accesses (DHA) in their Starbase/X11 Merge system to enable high performance when Starbase applications access the display [19, 18].

The remainder of the chapter is organized as follows. Section 6.2 gives a qualitative characterization of the workloads. Next, in Section 6.3, we analyze memory delays for X11 workload from three points of view: memory penalties by subsystem, cache effects, and TLB behavior. The chapter closes with a brief review of our major conclusions.

6.2. Workloads

We used the standard X11R5 distribution from MIT Athena. The X11 server was compiled with the default configuration, except that the PEX extension[14] was omitted. Table 6-1 describes the X11 clients used in this study. All X11 clients are written in C.

workload	Description	time
micro benchmarks		
destroy	window destruction, using *x11perf -repeat 5 -reps 10 -subs 10 100 -destroy*	2.6
resize	window resize, using *x11perf -repeat 2 -reps 5 -subs 10 100 -resize*	2.5
circulate	window circulate operations, using *x11perf -repeat 2 reps 5 subs 10 100 -circulate*	2.8
ftext	text painting, using *x11perf -repeat 5 reps 500 -ftext*	2.4
copy	bitmap copy, using *x11perf -repeat 5 reps 250 -copywinwin100*	11.4
scroll	window scrolling, using *x11perf -repeat 2 reps 250 -scroll500*	23.3
X11 clients		
splot	Splot is run four times on four different input files. Total size of splot input is 94K bytes.	12.4
gs	Ghostscript is used to preview a twenty page conference paper. Input file size is 251K bytes.	25.9
Other workloads		
gcc	The GNU C compiler converts a 17K (preprocessed) source file into optimized Sun3 assembly code. Not an X11 client.	3.7
compress	Data compression using Lempel-Ziv encoding. A 100K file is compressed then uncompressed.	1.3

Table 6-1: Experimental workloads.

Execution times are in seconds.

X11perf is distributed as a client program in X11R5. It is commonly used as a gauge of X11 server performance, measuring the time to repeat a given server operation some number of times. All the microbenchmarks for this chapter are runs of *x11perf* with different input parameters. *Splot* is a program for generating plots for PostScript and X11.

[14]PEX is the PHIGS extension to X11 used for three-dimensional graphics.

Ghostscript is an X11 previewer for the PostScript language from Adobe Systems. Version 2.6.1 of *gs* was used.

Two non-X11 clients from Chapter 4, *gcc* and *compress*, are included for comparisons between X11 clients and other workloads.

	instruction reads					data reads				data writes			
workload	idle	non-idle	%sys	%Xs	%Xc	non-idle	%sys	%Xs	%Xc	non-idle	%sys	%Xs	%Xc
destroy	0	33999	5.2	91.7	3.1	7128	5.5	91.1	3.4	4529	7.1	87.9	5.1
resize	0	35999	2.6	96.9	0.5	7382	2.2	97.2	0.6	2759	2.8	96.1	1.1
circulate	0	45999	2.9	96.7	0.4	9958	2.2	97.4	0.4	4284	3.4	95.9	0.6
ftext	0	62000	2.3	96.1	1.7	7921	4.2	92.5	3.3	4827	4.9	92.0	3.1
copy	0	92000	3.8	95.6	0.6	15912	3.4	95.5	1.1	14457	1.4	97.7	1.0
scroll	0	105999	2.0	97.9	0.1	32528	1.2	98.7	0.0	31740	0.5	99.5	0.0
splot	6910	148619	33.4	36.0	30.6	30852	32.2	34.9	32.9	16346	41.6	32.7	25.7
gs	3992	448018	10.0	30.8	59.2	93584	11.0	24.6	64.3	50994	14.6	33.4	52.0
gcc	63684	28899	21.0	0.0	79.0	5716	20.5	0.0	79.5	3810	28.6	0.0	71.4
compress	5555	16934	19.2	0.0	80.8	3249	18.9	0.0	81.1	2225	29.2	0.0	70.8

Table 6-2: Instruction and Data Reference Counts

This table shows event counts for each workload, along with the percentage contribution from the system (%sys), X11 server (%Xs), and X11 client (%Xc). The first column for each workload shows the number of idle instructions executed during that workload. All other counts and percentages are for non-idle events. All counts are in thousands.

Table 6-2 gives reference counts for the experimental workloads. The low percentage of kernel instructions for the microbenchmarks demonstrates that many types of X11 operations require relatively little kernel activity. Higher levels of kernel activity in *splot* and *gs* are attributed (at least partially) to the file I/O that these workloads require.

6.3. Experiments and Analysis

6.3.1. Memory Cycles Per Instruction

Figure 6-1 illustrates *MCPI* for the experimental workloads, as in figure 4-1. Contributions from different memory system components are indicated by shading, and system and user contributions are separated by a vertical bar.

For comparison, *MCPI* measurements for *gcc* and *compress* are also shown. Note that, unlike Figure 4-1, Figure 6-1 includes contributions from *user* uncached memory reads, as frame buffer reads and writes by the X11 server bypass the cache.

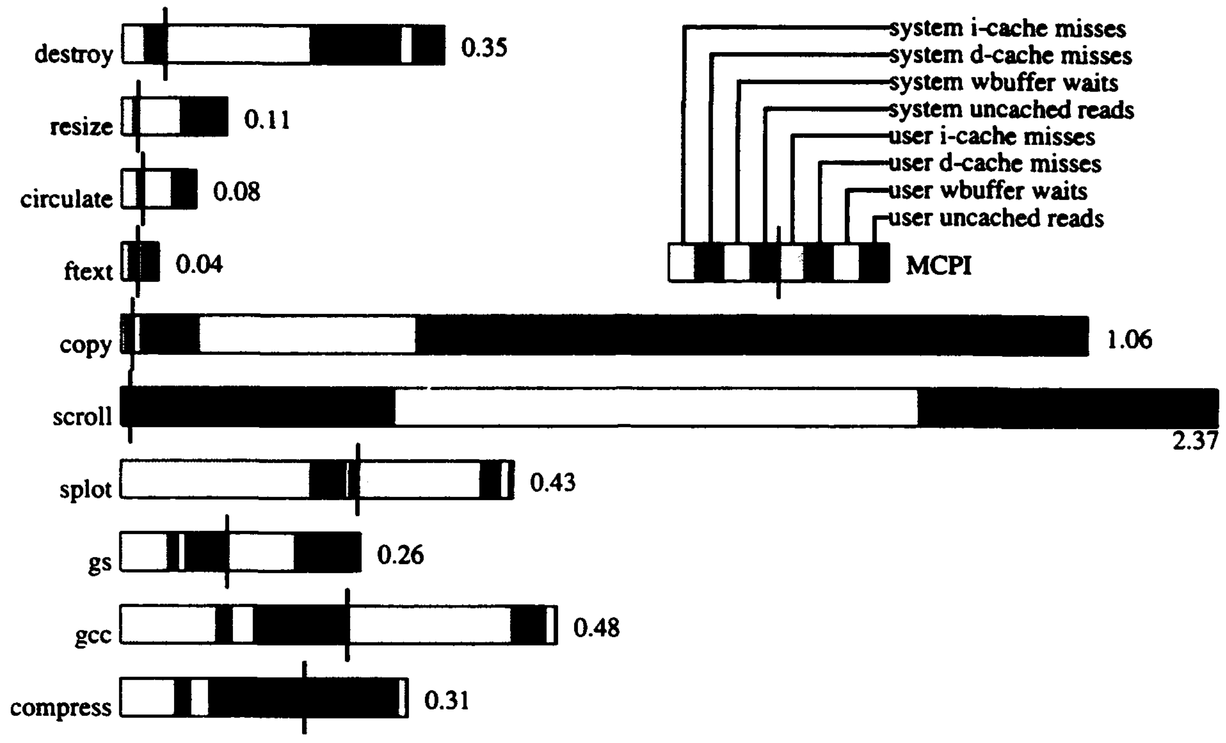

Figure 6-1: Baseline MCPI for X11 Workloads.

Each horizontal bar represents total *MCPI* for a given experimental workload, broken down between system/user and various components of the memory system. Contribution from system and user activity are separated by a vertical line. User activity includes X11 server and X11 client. The number at the right of each bar is the *MCPI* for that workload. Startup and shutdown effects were excluded by omitting several million instructions at the beginning and end of each simulation experiment. *MCPI* for *gcc* and *compress* are included for comparison with workloads that are not X11 clients.

As can be seen from Table 6-2, operating system overhead is low for the six microbenchmarks. This is reflected in Figure 6-1 as low system *MCPI*. With the X11 server accessing the frame buffer directly, kernel activity during the microbenchmarks is dominated by TLB faults and socket communication, both of which are relatively inexpensive as compared to the activity of disk I/O intensive workloads. In contrast, user-level *MCPI* varies significantly across the microbenchmarks, and is dependent on server activity. The worst behavior occurs for *copy* and *scroll*, which incur substantial write-buffer and uncached-read penalties due to a high density of frame-buffer references.

Comparing system behavior for the microbenchmarks to that of *splot* and *gs* shows that there is substantial additional system overhead for the realistic benchmarks. This reflects greater variation in system activity due to the addition of file I/O and is consistent with behavior observed for system-intensive and I/O intensive applications such as *gcc*.

Turning to user-level overhead, Figure 6-1 shows that many X11 clients have significant user instruction cache *MCPI* contributions, sometimes higher than system i-cache *MCPI* contributions. This is unusual for integer workloads[15]. In the next section we discuss how competition within and between address spaces contributes to poor instruction cache behavior for both system and user.

Penalties from the write-buffer and uncached memory reads appear problematic in microbenchmarks but aren't significant in more realistic workloads. For frame-buffer writes, the X11 server benefits from the combination of write-buffer and writes through the uncached segment. Together they permit frame buffer writes to proceed at top speed without disturbing the contents of the data cache.

6.3.2. Cache Effects

We consider two types of cache misses:

- *Inter-Context Competition* occurs when references from two or more active address spaces displace each other in the cache. Client-server systems such as X11 introduce a user level server context in addition to the application and system context of a workload such as *gcc*. The additional server context could induce more competition in the cache.
- *Self-Interference* misses occur when two active instructions in the same address space collide in the cache. The Ultrix page-mapping algorithm ensures that self-interference misses will not occur within an address space if a program's text is smaller than the cache size. Many of the SPECmarks have text smaller than the cache, but the text of X11 server, *gs* and *splot* are all much larger. With localities spread throughout large text, self-interference misses are more likely to occur.

We discuss inter-context competition first.

6.3.2.1. Inter-Context Competition

An X11 workload is composed of three contexts: X11 client, X11 server, and the operating system. Inter-context competition occurs when two or more contexts compete for memory resources (such as cache lines), causing degraded performance. To simulate a system without competition, we ran experiments with memory reference trace from only one source (eg. use trace from the kernel only), then summed events over all three

[15] *gcc* is exceptional in this respect; see Chapter 4.

runs. The sum represents the behavior of a hypothetical machine where each context has its own private memory system, hence a system where all competition has been eliminated. Figure 6-2 compares *MCPI* for several workloads with and without competition.

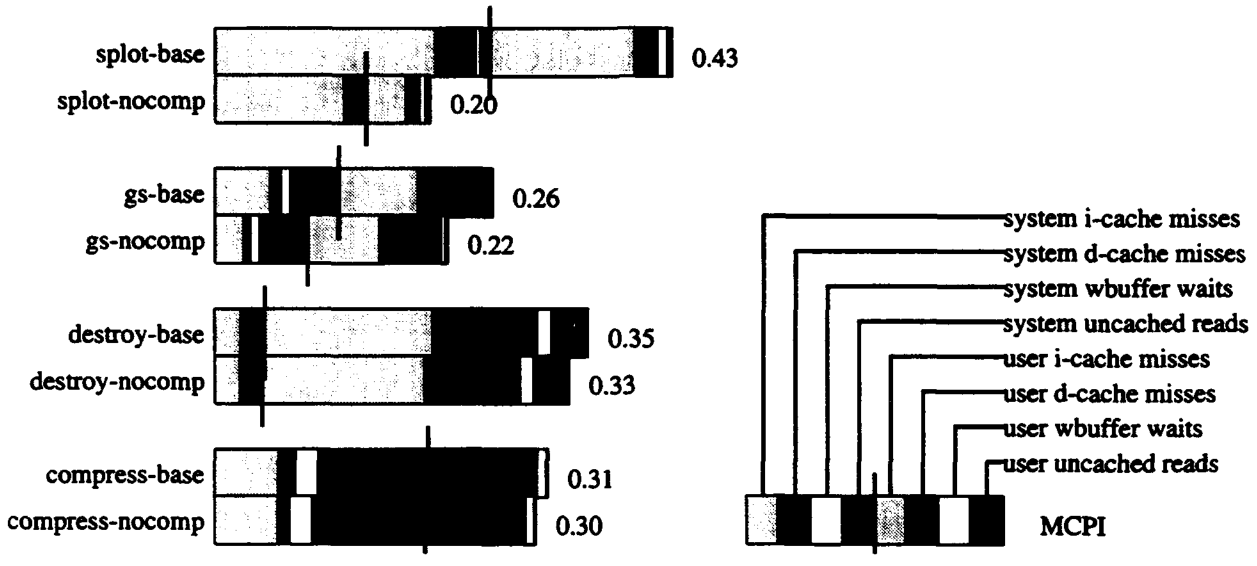

Figure 6-2: *MCPI* with and without inter-context competition.

This figure shows the effect of inter-context competition on memory system performance. Each horizontal bar represents total *MCPI* for a given experimental run, as in Figure 6-1. We show two bars for each workload, the upper corresponding to the base system, and the lower representing a system in which competition between address spaces has been eliminated by giving each address space a private memory system. This figure shows that inter-context competition has major impact on instruction cache behavior for realistic X11 workloads.

Figure 6-2 shows that, for realistic X11 workloads, inter-context competition has significant impact on overall *MCPI*. For both *splot* and *gs*, competition is responsible for a large proportion of system i-cache misses. Additionally, eliminating competition from *splot* causes a drastic reduction in user i-cache miss rates. The instruction cache requirements of *destroy* microbenchmark are modest, so eliminating competition has little effect. Similarly, compress shows little benefit when competition is eliminated.

We compared missing instruction addresses in the X11 server to counts of kernel instruction references for a run of *splot* to identify probable inter-context conflicts. An example of such a conflict was between the Ultrix general exception handler, `exception()`, and the X11 server memory allocation routine, `malloc()`. Both routines are called frequently and are located in such a way that they overlap on a 4K byte memory page. Kernel text pages are not mapped, so `exception()` will always be located in the same place in the cache. User text is mapped, so the location of `malloc()` in the cache will depend on the virtual-to-physical page assignment. In Ultrix the page assignment is a function of virtual page number and process ID. Given that the address space identifier for the X11 server is random, that the cache is direct-mapped, and that the cache size is sixteen pages, the probability that the above conflict will occur is 1/16. Many such conflicts were identified for the *splot* run. The large working sets of X11 workloads combined with frequent mandatory context switches makes competition misses more likely.

These results confirm earlier research on cache competition which demonstrated significant penalties for warming up the cache after a context switch [57]. The earlier study found competition to be important in multitasking and compute-bound workloads, but a non-issue in the client-server workload they tested. However, the earlier study did not include system behavior and used a synthetic client-server workload dominated by communication, a workload more similar to the microbenchmarks used for this study than the realistic workloads. Our work complements the prior study by including system effects in the memory reference stream, and by demonstrating that inter-context competition can also have impact for client-server workloads.

6.3.2.2. Self-Interference misses

We measured the impact of self-interference misses by replacing the direct-mapped caches in the simulator with two-way set associative caches of the same size. Figure 6-3 illustrates the combined effects of competition and associativity on *MCPI* for *splot* and *gs*. To isolate the impact of self-interference misses from that of competition misses, compare runs where all competition misses are eliminated (*bench*-nocomp and *bench*-a+nocomp). In this comparison, all misses eliminated by associativity are from conflicts within an address space. For both *gs* and *splot*, associativity eliminates a significant number of self-interference misses.

Figure 6-3 also shows the effect of associativity on inter-context competition. The two-way set associative caches eliminate some inter-context competition, but not all (compare *bench*-assoc and *bench*-a+nocomp).

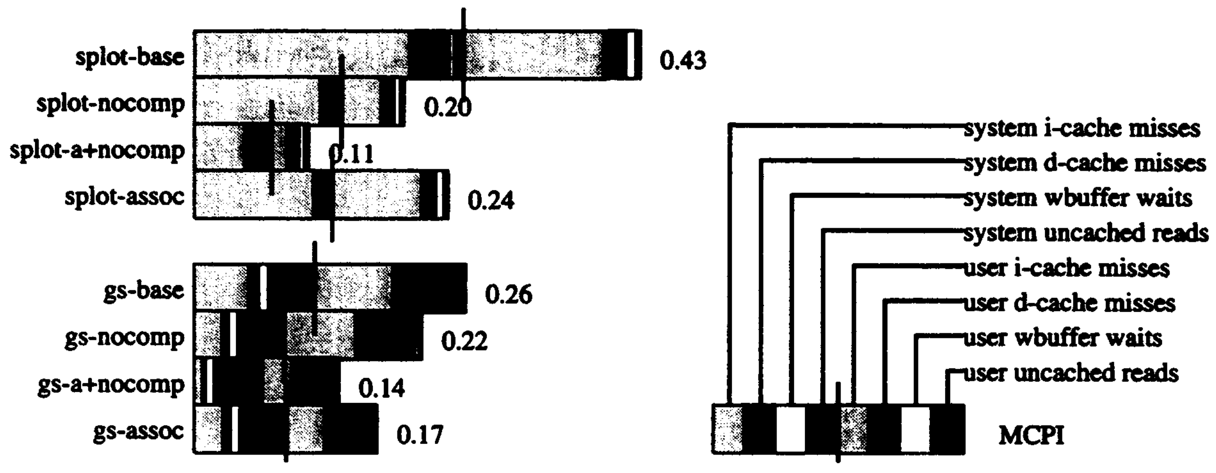

Figure 6-3: Inter-Context Competition, Associativity, and *MCPI*.

This figure shows the effect competition and self-interference misses on memory system performance. Each horizontal bar represents total *MCPI* for a given experimental run, as in Figure 6-1. We show four bars for each workload: the base system (*bench*-base), without competition (*bench*-nocomp), with associativity and without competition (*bench*-a+nocomp), and with associativity (*bench*-assoc). This figure shows that both competition and self-interference misses have significant impact on memory system behavior for realistic X11 workloads, particularly in the instruction cache.

6.3.2.3. Summary

For the two realistic X11 workloads we consider, both inter-context competition and self-interference misses have significant impact on memory system behavior, with the most significant effects occurring in the instruction cache. Strategies have been described [55] for avoiding text conflicts within an address space, but it is difficult to envision a practical software system to avoid competition between address spaces. The problem could potentially be addressed in hardware with cache associativity, although this could increase machine cycle time or hardware cost. Many current systems rely on good luck to avoid inter-context competition. As the gap between CPU and memory speed grows, and as users demand improved performance for interactive and multi-address space systems, more aggressive cache designs may be required. The lack of client/server workloads in standard benchmark suites could lead hardware developers to believe that competition between user-level contexts is a non-issue. Our measurements suggest it deserves more attention. Recent work on dynamic page mapping policies [14] shows promise in addressing this problem.

6.3.3. TLB Behavior

workload	user	system
splot	1.58	0.28
gs	1.49	0.14
gcc	1.11	0.07

Table 6-3: TLB Misses per 1000 instructions.

System TLB misses include misses to both user and system segments.

Table 6-3 shows TLB miss data for *splot*, *gs*, and *gcc*. Compared to other integer workloads, X11 applications have poor TLB behavior. Both *splot* and *gs* show significantly higher miss rates than *gcc*, which is relatively demanding among integer workloads [22]. Three phenomena contribute to increased TLB miss rates:

- The X11 server needs over 200 page mappings to address the entire frame buffer. Any operation that paints a significant part of the screen will tend to flush the TLB.
- The X11 server, *gs*, and *splot* all have relatively large program text. This increases the likelihood that localities will be spread across multiple text pages, which in turn increases the demand on TLB resources.
- X11 applications involve two interacting user contexts, as opposed to one context for *gcc*. Multiple contexts mean more fragmentation and increased competition for limited TLB resources.

During the run of *splot*, 280000 user TLB misses occurred. There were about 680000 during *gs*. Estimating 20 cycles to service a TLB miss [60], the penalty for TLB faults is less than 0.04 CPI for both workloads. Thus, the impact on overall performance is not significant for the memory system we simulated.

The impact of the TLB on overall performance is dependent on the performance balance of memory system components. Processors such as the DEC Alpha 21064 [5] rely on fewer TLB entries with larger and oversized pages to achieve good TLB behavior. If software systems such as X11 don't make good use of these new features, miss rates will go up, increasing the impact of the TLB on overall performance. Also, the penalty for a single TLB miss could increase. An earlier study used 100 cycles as an estimate of the TLB miss penalty for a futuristic machine [23]. As the balance of TLB to cache resources changes, TLB performance could become an important issue. X11 workloads require more TLB resources than popular benchmarks such as *gcc*. If computer systems are designed to optimize the performance of the popular benchmarks, systems such as X11 can be expected to suffer.

Our measurements show degraded TLB behavior for X11 workloads as compared to other integer codes. Researchers at the University of Michigan [60] measured similar TLB behavior for the Mach 3.0 operating system [1, 41]. The two independent studies suggest a broader conclusion — that page behavior for user-level client/server systems induces substantially elevated TLB miss rates.

As a final note, competition also affects TLB behavior. In our measurements, competition accounted for 60% of user TLB misses in *splot* and 30% of user TLB misses in *gs*. Note that additional associativity cannot help here, as the DECstation 5000/200 TLB is already fully associative.

6.4. Conclusions

Memory system behavior for X11 workloads differs significantly from the batch-mode programs typically used in memory system studies. With large program text, a large mapped frame buffer, and multiple competing contexts, X11 workloads can present a far greater load on instruction caches and TLBs than typical throughput benchmarks.

Cache associativity and TLB size are sensitive issues for hardware designers. For many machines, increasing these parameters would have direct impact on machine cycle time, the principle metric driving performance improvements in microprocessors. As machine cycle time dominates performance for many current benchmarks, there is a potential conflict between high throughput for benchmarks and low latency for large client/server systems. Optimal throughput and optimal latency may not be possible in the same machine.

Our measurements are specific to X11 clients and server, but similar behavior can be expected in other client/server systems. Instruction cache and TLB behavior is aggravated by large user text, frequent and mandatory context switches, and multiple active contexts. Any system with these characteristic will probably have similar behavior. In systems with multiple heavy-weight servers, contention for memory system resources will be even more intense.

X11 workloads are larger and more complex than the standard workloads used in computer performance evaluation. These distinctions suggest similarities between X11 workloads and operating system activity. This similarity is also reflected in the memory system behavior of X11 workloads. Workloads as complex as X11 applications are rare in workstation performance evaluation. However, popular consumer software systems like spreadsheets and document processing tools are even larger and more complex. If the performance of these workloads is important then they ought to be the standard case for performance evaluation and not something extreme or unusual. If such workloads were used as benchmarks, hardware designers could focus their attention on the requirements of these workloads for efficient execution. As computer hardware and programming methodologies adapt to improve the performance of large complex user code, system code should also benefit.

Chapter 7
TLB Behavior

As an example of how the address tracing system can be applied to problems in hardware design, this chapter presents the results of a simulation-based study of various translation lookaside buffer (TLB) architectures in the context of a modern RISC processor. We studied the performance of two-level and fully associative TLBs. We found that the dominant factor in determining behavior was the amount of memory mapped by the TLB. A small first-level FIFO instruction TLBs can be effective in two-level TLB configurations. In a machine model where most operating system text and data is referenced with little TLB overhead, system effects generally cause a reduction in the TLB miss rate, as user-level miss activity is amortized over a larger number of instructions.

This is an extended version of an earlier study [23] which used similar analysis of TLB behavior but excluded system behavior; the following includes system behavior. Both Mach 3.0 and Ultrix traces were used for this study. An independent study conducted at the University of Michigan also considers the impact of system references on TLB behavior for Ultrix and Mach 3.0, although with a different simulation environment and different user workloads [60]. The results of our experiments are consistent with the earlier results.

7.1. Introduction

In computer systems with virtual memory, a TLB is typically used to provide fast translation of virtual addresses generated by instruction execution to physical addresses needed for cache tag comparisons. Both physically and virtually addressed caches require address translation. With physically addressed caches, the TLB lookup is in the critical path of cache access, so low latency and miss rates are crucial for memory system performance. The TLB is a cache, speeding up access to entries in the page table, where complete information on virtual to physical memory mappings is maintained. Most modern machines use split instruction and data caches, and this configuration is assumed (unless stated otherwise) in the remainder of this chapter. Given this context, we consider several possibilities for the TLB implementation:

- **A single TLB shared between instruction and data caches.** To reduce contention, the TLB can be dual-ported. This introduces complex circuitry, doubling the size of the TLB without increasing its capacity.
- **Independent TLBs for instruction and data caches.** The instruction TLB should be made smaller than the data TLB, as instruction reference streams exhibit greater locality than those for data. The appropriate size tradeoff is difficult to determine and once made is fixed. If the instruction TLB is too small, performance will suffer. If it is too large, the space available for the data TLB is compromised and again performance suffers.
- **Two-level TLB architectures.** A small instruction TLB (i.e., *micro-TLB*) can be refilled from a larger single-ported shared TLB, primarily used for data references. This option is described in more detail below.

A micro-TLB is a fully associative TLB with a very small number of entries (probably less than eight) which is reloaded in hardware from a larger shared TLB. A number of recent machines use micro-TLBs, including the MIPS R4000 [56], though they are invisible at the architecture level. A micro-TLB is accessed in parallel with the instruction cache. On a miss, the micro-TLB is reloaded from the shared TLB. As the larger TLB is single ported, the CPU may stall for a few cycles with data references suspended while the TLB is busy, but this penalty is much less expensive than that of a full TLB miss. We assume 3 cycles as the micro-TLB miss penalty in this chapter. Because of the high locality in instruction reference streams, and the relatively small miss penalty, acceptable miss rates can be achieved with a small micro-TLB. The balance of instruction and data entries in the shared TLB is determined dynamically, unlike in the second option above. In addition the shared TLB need not be dual-ported, so the extra space can be used to increase its capacity.

Because the TLB can be in the critical path of memory access, good TLB performance is essential to good overall performance of a machine. TLB design has been complicated in several recent architectures with split instruction and data TLBs. To date, such designs have received negligible attention in the research literature. We present experimental results to characterize the behavior of split as well as shared TLBs.

Another feature found in several recent architectures is TLB entries that can map variable size pages. When such a TLB entry is loaded with a new mapping, it is also loaded with the size of the page to be mapped. Typically, the size is restricted to a power of two and may range from 4K bytes to a gigabyte [33, 56]. Although there are several obvious applications of variable size pages, such as mapping operating system text and graphics frame buffers, it is not yet understood to what degree they can be used to improve the execution rate of application code. Applications performance for contiguous references could improve by accessing the segment as a single large page mapped by a single TLB entry. Applications which scatter references across a sparse address space have little hope of benefiting from large pages without significantly increased memory usage. Address traces and reference counting tools are useful for recording dynamic patterns of memory access to aid in understanding the applicability of these structures.

The many recent studies of memory system behavior and performance concentrate almost exclusively on cache design [74, 66]. Less attention has been given to TLB performance. Early studies showed that TLB miss penalties can consume as much as 6% of all machine cycles [25] and 4% of execution time [26], and hence can have a significant impact on machine performance. However, these results were for VAX computers with 512 byte page sizes — an order of magnitude smaller than is typical today — and main memory sizes two orders of magnitude smaller than those considered in this study.

Wood [87, 86] proposed in-cache address translation as an alternative to a TLB. His work has shown that such methods are effective for programs such as Lisp applications and operating systems, where the working set is spread over a large address space. They are less useful for behavior such as is seen with typical C programs, where memory activity is concentrated in the bottom of several segments. His methods also become less applicable when memory access times become large with respect to processor speed.

Finally, previous TLB studies [26, 86] have considered set-associative or direct-mapped organizations. These were common when TLBs were made from discrete MSI and LSI RAMs. Recently, however, VLSI RISC microprocessors (e.g., [33, 56]) typically make use of fully-associative TLBs, since these require about the same area as set-associative TLBs when implemented within a VLSI chip. Thus the TLB implementations studied in this chapter are all fully-associative designs.

This chapter is concerned with TLB performance. Understanding the relation between TLB performance and overall machine performance is a different question, involving the balance of compulsory to capacity misses and the relative ability of caches and TLBs to map multiple localities. The first access to a page results in a *Compulsory* TLB miss. In this situation, cache misses also occur, the cost of which might overshadow the TLB penalties. Other TLB misses are *capacity* misses, when a program returns to a locality that has been replaced out of the TLB even though it might still be present in the cache. This phenomena becomes more common as cache sizes increase. In this chapter, CPI effects of TLB performance are discussed assuming no delay for memory system components other than the TLB. This can be deceptive, as such a treatment trivializes the question of cache performance. More exact results require simulation of a complete memory system.

The remainder of the chapter is structured as follows. First we give some brief details on our experimental workloads and methodology. This is followed by a discussion of instruction TLB behavior, both for micro-TLBs and for independent instruction TLBs. Data and shared TLB results are presented next, followed by a discussion of variable size pages, and finally some concluding notes.

7.1.1. Workloads and Methodology

We used a variety of workloads in simulating the various TLB configurations, including several SPECmarks, plus other workloads meant to anticipate more demanding workloads. *Tree* is a recursive, data intensive benchmark written in C-Scheme [11]. *Magic* [62] is a VLSI layout tool. In this run it was extracting the MultiTitan CPU chip [44]. We also used a multi-tasking workload, running the following programs:

- *gcc*
- *magic* extracting the MultiTitan CPU chip
- *ld* loading *magic*
- *tree* with a 10 megabyte heap
- a loop running the shell programs *cp*, *cat*, *sed*, *ls*, *ps*, and *rm*.

Short running programs were put in loops, so that their execution would continue throughout the entire run. This mix is is meant to be comparable to the mix used in previous trace-based studies by Borg et al. [16].

Experimental runs from the original set of experiments do not include traces of system activity. These runs were supplemented with runs that include system traces, using the same set of workloads as the Ultrix vs Mach 3.0 comparison. In the presentation of material for each type of TLB, we start by discussing experiments with user behavior only. This provides the best context for distinguishing between the different behaviors seen in user activity. After discussing user activity we consider system activity and its influence on overall performance. The experiments which include system activity are presented in separate tables to permit a comparison of behavior with and without system activity.

Our model of system interaction with the TLB is based roughly on the DECstation/MIPS system architecture. We assume that kernel text and most kernel data is unmapped. This assumption is also appropriate for machines where kernel memory is mapped but without significant TLB miss activity, as would occur with large TLB entries. For Ultrix, this means that almost all system activity is isolated from the TLB. For Mach 3.0, most kernel activity is isolated from the TLB. We assume that the UNIX server and emulation library are mapped and do induce TLB misses.

7.2. Instruction TLB results

Instruction reference streams place lesser demands on TLB resources than data reference streams. Instruction references generally exhibit higher locality, both spatial and temporal. Also there is generally less memory involved. The largest text segment of the SPECmarks, when compiled for the DECstation, is the Gnu C compiler *gcc* with 688K bytes. The average text segment size is around 200K bytes. Data segments are frequently much larger. Data references for *nasa7*, a benchmark for numeric computation, range over a space of over three megabytes.

Because of the different performance characteristics of TLBs and micro-TLBs, they are discussed separately.

7.2.1. Micro-TLBs

Micro-TLBs were simulated with sizes varying from one to eight entries and *Least Recently Used* (LRU) or *Least Recently Replaced* (FIFO) replacement algorithms. Page sizes of 4K and 16K bytes were used for the simulations. We first discuss behavior with system activity excluded. This simplifies the discussion of behavior and emphasizes differences between workloads. At the end of the section we extend these results for the impact of system effects on Mach 3.0 and Ultrix.

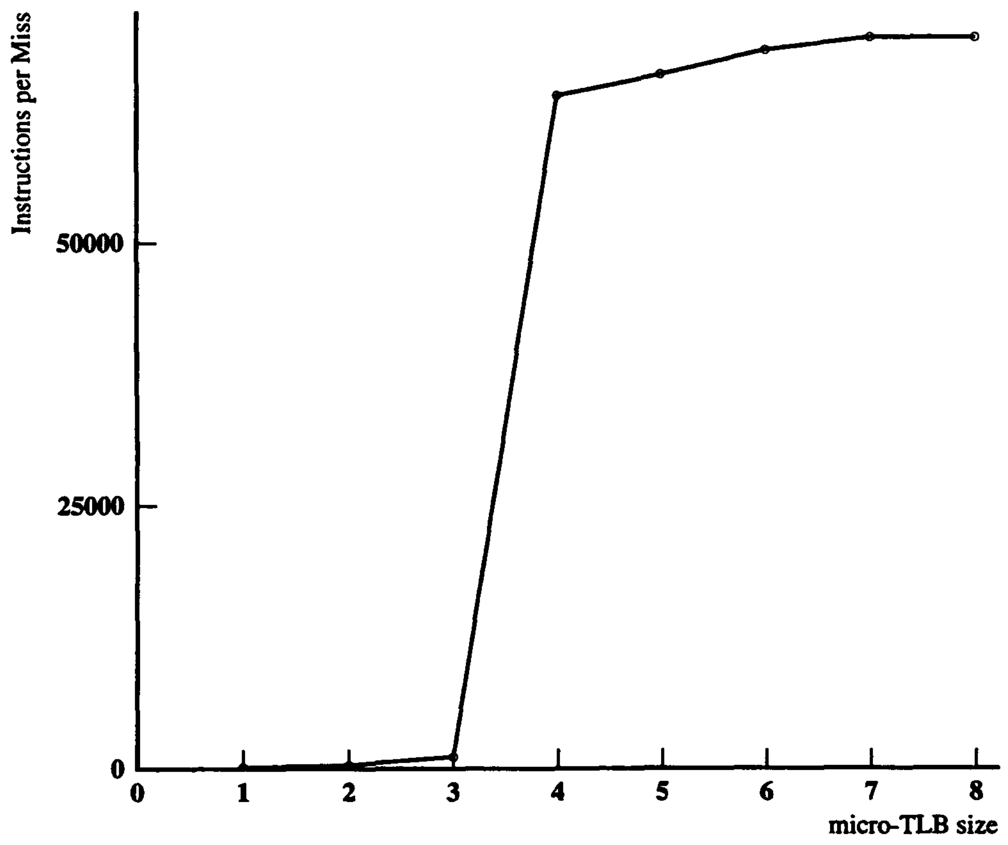

Figure 7-1: *Eqntott* micro-TLB behavior

This figure illustrates micro-TLB behavior for the *eqntott* workload across a range of micro-TLB sizes. System activity is not included. The micro-TLBs use FIFO replacement and 4K byte pages. Three domains of behavior can be identified in this figure, thrashing with the smaller micro-TLBs, improvement as the number of entries approaches working set size, and stable good performance when the working set size has been reached.

Figure 7-1 is a plot of the simulation results for the *eqntott* benchmark, illustrative of micro-TLB behavior. The number of entries in the micro-TLB varies along the *x* axis. The *y* axis is scaled in instructions per miss, the reciprocal of the miss rate. We use plots of instructions per miss because they illustrate interesting behavior more clearly than plots of miss rate. Data points corresponding to good performance are towards the top of the graph, those for poor performance are toward the bottom, and the points are spaced in a meaningful way rather than disappearing along the X axis.

There are three general regimes of behavior to be observed. Notice the flatness of the curve on the left side of the graph. In this region, thrashing is occurr he number of micro-TLB entries is well under the number of instruction pages in the .king set of the program. Consequently, micro-TLB performance is relatively poor: 442 instructions per miss with two micro-TLB lines, 1490 instructions per miss with three.

After the flat part of the curve comes a region where performance improves rapidly. For *eqntott* this occurs between three and four micro-TLB entries. Lastly comes another relatively flat region, where the micro-TLB has enough entries to map the entire working set of the program. In this region, additional micro-TLB entries do little to improve performance.

One issue which turned out to be uninteresting is the effect of context switches on the micro-TLB. We considered two possible models of behavior, a *pessimistic* model in which the contents of the micro-TLB is flushed after every context switch, and an *optimistic* model where the contents of the micro-TLB is preserved across context switches. Our experiments showed that the overall impact of the pessimistic model is not important. It has measurable impact for workloads where TLB behavior is very good, increasing the number of compulsory misses, but even with this increase behavior remains very good. When TLB behavior is bad, most misses are due to capacity problems. These misses are much more frequent than compulsory misses, and dominate behavior for both the pessimistic and optimistic model. This is consistent with the results in Section 4.3.3, which showed that context switches were not frequent enough to have a significant performance impact. The simulations used to generate data for this chapter use the pessimistic model.

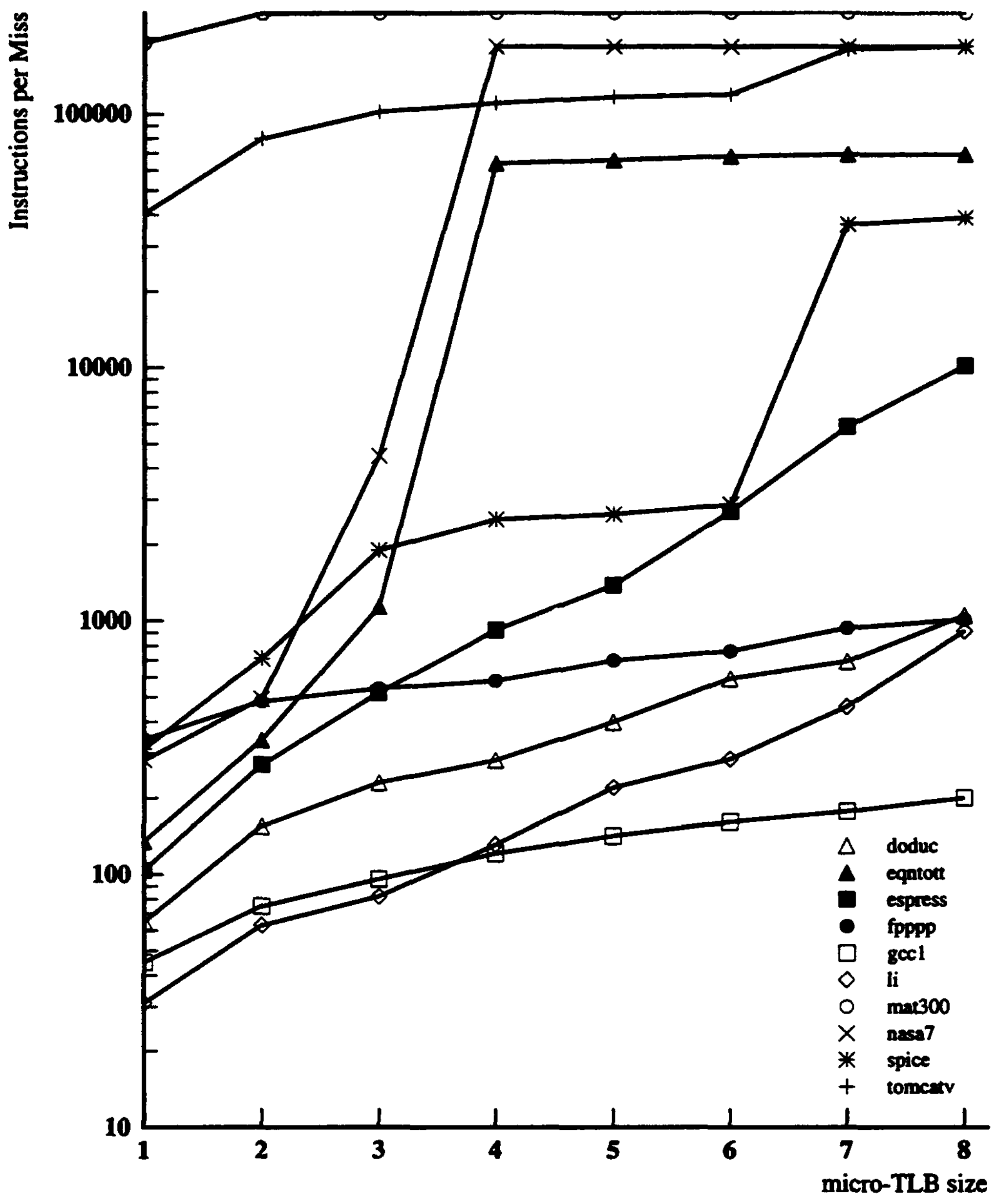

Figure 7-2: Micro-TLB Behavior, 4K byte page

This figure shows experimental results for micro-TLB simulations using FIFO replacement. System activity is not included. Note that the *y* axis uses a log scale. For a given workload, the micro-TLB working set size can be identified by noting where the corresponding plot reaches a plateau.

Figure 7-2 illustrates micro-TLB performance for the SPECmarks. Notice a log scale is used for the *y* axis of this graph, while the *y* axis in Figure 7-1 used a linear scale. Although the log scale tends to obscure the different domains of behavior, it makes it possible to compare all the SPECmarks on the same graph. If a micro-TLB miss penalty of 3 cycles is assumed and the average number of instructions per miss is 333 or less, then about one machine cycle per one hundred instructions (i.e. 0.01 cycles per instruction - CPI) is lost to micro-TLB misses. With one micro-TLB entry, more than half the SPECmarks have this much of a penalty. With a two entry micro-TLB, 40% are at this penalty level.

Four of the SPECmarks have mediocre micro-TLB performance. *Fpppp* and *doduc* are both floating-point benchmarks. Both of them are noted for their poor instruction cache performance, predictive of the observed micro-TLB behavior. The miss rate for *fpppp* for a simulated 4K byte instruction cache with 16 byte lines is 23%, due to its long basic blocks (an average of 130 instructions per branch for the entire run). Computation in *doduc* is spread over a large number of procedures. Simulations show it has an miss rate of 11% for a 4K byte cache instruction cache.

Two language processing programs, *gcc* (the Gnu C compiler) and *lisp* (a lisp interpreter), have bad micro-TLB behavior. The I-cache miss rates for *gcc* and *lisp* are 10% and 2%, respectively. Their micro-TLB behavior is explained in considering the structure of the programs. For example, *Gcc* has a large amount of code and it tends to make many nested procedure calls. We believe that the observed behavior results from there being eight or more procedures involved in most of *gcc*'s localities, and that these procedures tend to be spread over more than eight pages.

The SPECmarks were also simulated for micro-TLBs using a 16K byte instruction page size. The results are shown in Figure 7-3. At this page size, with 7 micro-TLB entries, the working set of virtually all the programs appears to have been reached, with the exception of *gcc*. The amount of memory fragmentation induced by the change to 16K byte pages can be inferred from the change in TLB resource demands of the programs. For example, *eqntott* uses $4 \times 4K = 16K$ bytes of instruction memory with 4K pages, and 48K bytes with 16k pages. Spice grows from 28K to 64K bytes. Fragmentation for 16K byte pages could be reduced with compilers and loaders that used heuristics or feedback information to relocate the most active code to make it adjacent in memory.

Another micro-TLB design parameter that was considered is replacement policy. For two entries, LRU is easily implemented in hardware. For more than two entries,

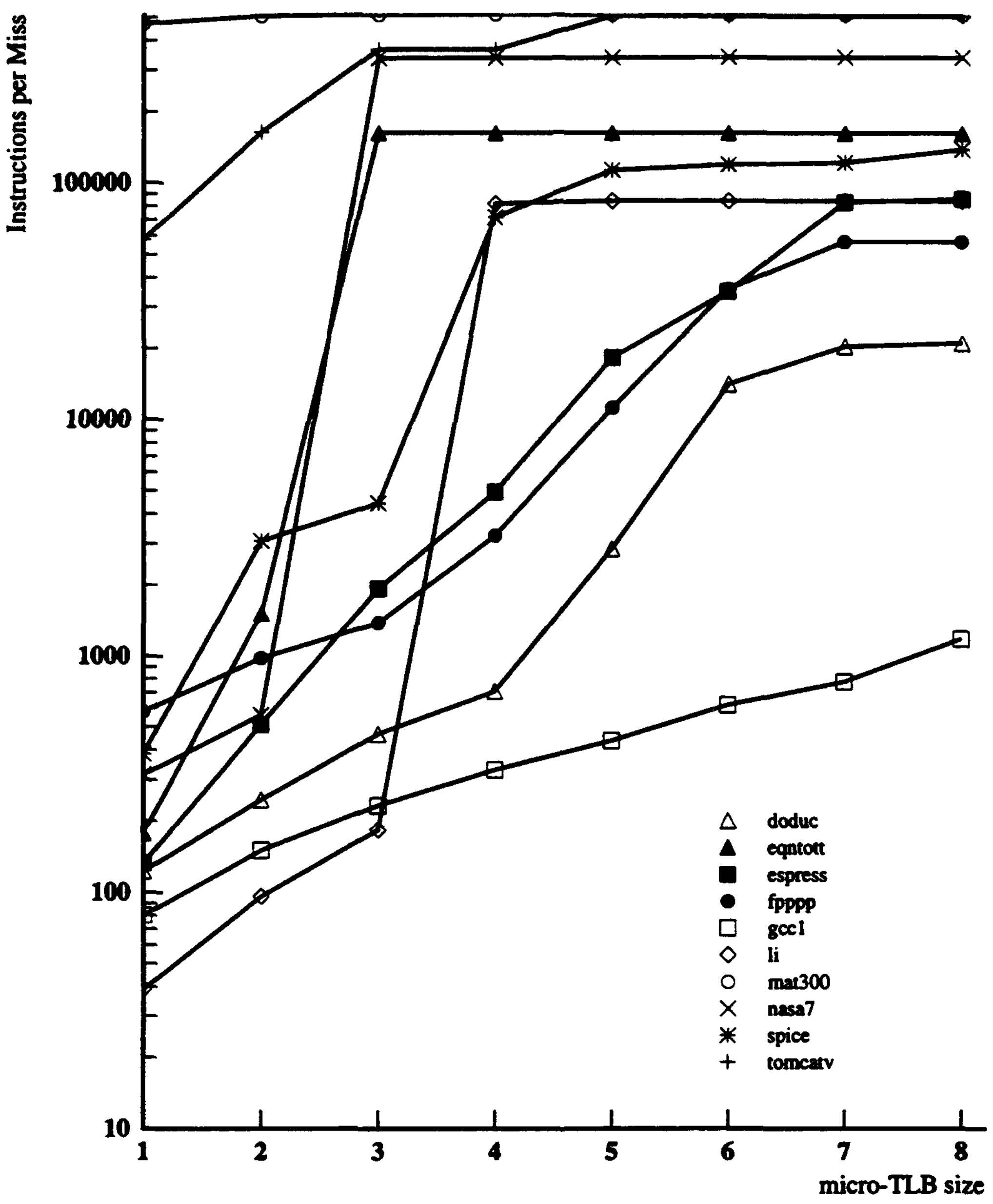

Figure 7-3: SPECmark micro-TLB Behavior, 16K byte pages

These experiments are similar to those of Figure 7-2, except that 16K byte pages rather than 4K byte pages were used. System activity is not included. Note that several workloads reach their working set sizes with 16K byte pages that did not with 4K byte pages.

hardware LRU becomes more difficult. An interesting alternative for micro-TLBs is *least recently replaced.* This has the advantage of a relatively straightforward hardware implementation as a first-in first-out queue. *FIFO* is used in this exposition to refer to this replacement policy.

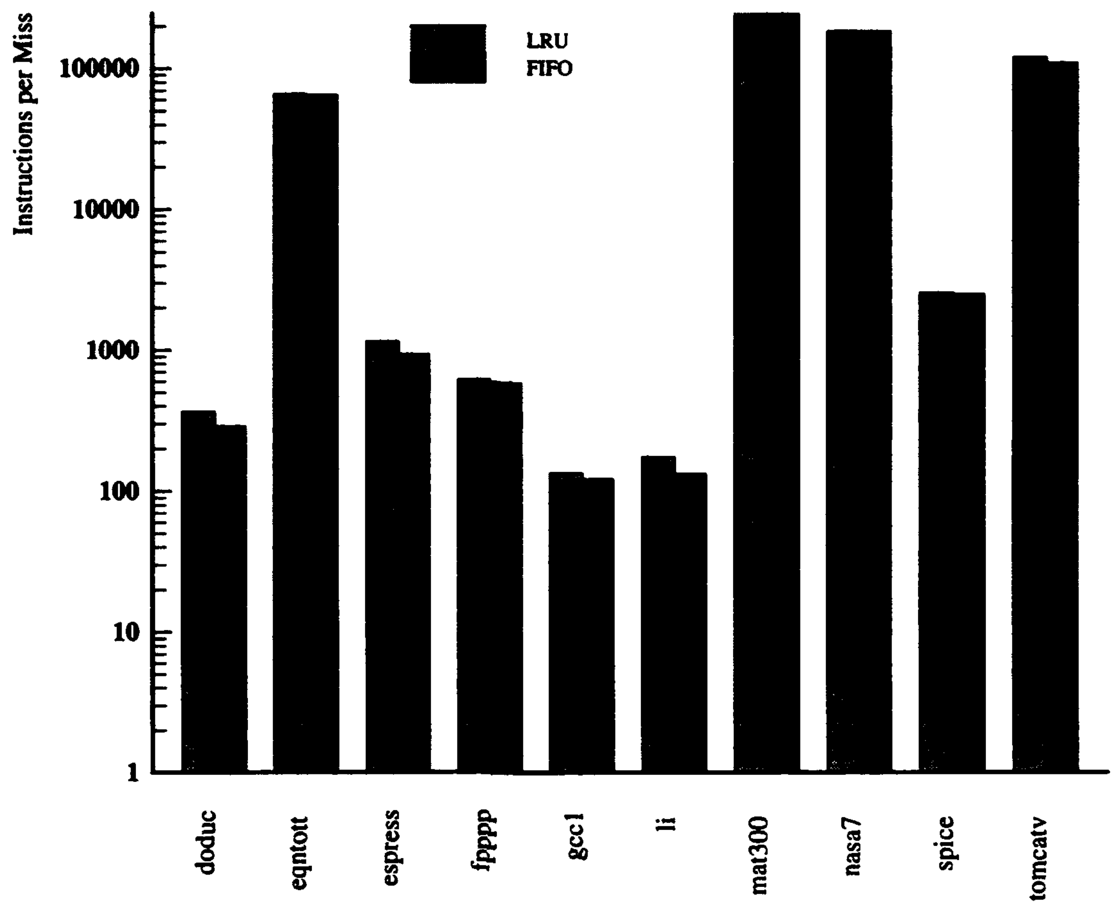

Figure 7-4: LRU vs. FIFO Replacement

This figure shows the effect of micro-TLB replacement policy, for LRU and FIFO replacement. The experiments used a 4 entry micro-TLB and 4K byte pages. System activity is not included. This figure shows that a FIFO replacement policy has comparable performance to an LRU policy. FIFO has the advantage of a straightforward hardware implementation.

Figure 7-4 shows relative performance of LRU and FIFO replacement policies for ten SPECmarks. LRU is uniformly better, but not by a large amount. Again, the log scale makes it possible to compare all the benchmarks in the same graph, although it tends to obscure the real difference in performance, which is sometimes as much as a factor of two. Nonetheless, FIFO performance is always comparable with LRU performance when the overall affect on CPI is considered. This means FIFO is an interesting replacement policy for micro-TLBs of size greater than two.

Simulations were also run to compare LRU, FIFO and Random replacement policies in full size data TLBs. It was found that FIFO performs uniformly better than Random, failing only in pathological worst case situations. For most machines with hardware to support Random replacement, FIFO can be easily implemented, although it is expected that Random replacement will be used to avoid pathologic behavior.

	1×4KB pg	2×4KB pg	4×4KB pg	8×4KB pg	1×16KB pg	2×16KB pg	4×16KB pg	8×16KB pg
doduc	**0.015415**	**0.006433**	**0.003552**	0.000953	**0.008111**	**0.004078**	0.001416	0.000047
eqntott	**0.007415**	0.002951	0.000016	0.000014	**0.005647**	0.000661	0.000006	0.000006
espress	**0.009610**	**0.003687**	0.001089	0.000099	**0.007500**	0.001966	0.000202	0.000012
fpppp	0.002936	0.002085	0.001727	0.000988	0.001723	0.001022	0.000309	0.000018
gcc	**0.021989**	**0.013269**	**0.008288**	**0.005010**	**0.012539**	**0.006648**	0.003042	0.000844
lisp	**0.031792**	**0.015781**	**0.007661**	0.001097	**0.025473**	**0.010456**	0.000012	0.000012
mat300	0.000005	0.000004	0.000004	0.000004	0.000002	0.000002	0.000002	0.000002
nasa7	**0.003544**	0.002037	0.000005	0.000005	0.003180	0.001788	0.000003	0.000003
spice	0.003199	0.001407	0.000399	0.000026	0.002598	0.000326	0.000014	0.000007
tomcatv	0.000025	0.000013	0.000009	0.000005	0.000017	0.000006	0.000002	0.000002
ccom	**0.030306**	**0.016114**	**0.007844**	**0.004098**	**0.017705**	**0.008653**	0.001761	0.000174
sed	**0.033778**	**0.008355**	0.000070	0.000064	**0.009710**	0.000031	0.000031	0.000031
tree	**0.074999**	**0.034553**	**0.017515**	0.001264	**0.062228**	**0.021160**	0.002427	0.000215
magic	**0.014811**	**0.007654**	0.002853	0.000606	**0.013080**	**0.006197**	0.001599	0.000287

Table 7-1: SPECmark FIFO micro-TLB Miss Rates

Experiments for this table exclude system references.

Table 7-1 shows miss rates for a selection of workloads. The experiments for Table 7-1 used user references only. Under the assumption of a 3 cycle miss penalty, figures in bold-face indicate where the micro-TLB penalty is greater than 0.01 CPI. Again, with 4K byte pages, most of the SPECmarks achieve reasonable performance with a two entry micro-TLB, and the others show improvement for larger micro-TLB sizes. Figure 7-5 shows how data from Table 7-1 can be used to estimate CPI contribution for a given SPECmark, micro-TLB configuration, and miss penalty.

2 entry FIFO micro-TLB with 4K byte page
3 cycle miss penalty

CPI Contribution = 3 * 0.0133 = 0.0399 CPI

Figure 7-5: Estimating micro-TLB CPI Contribution for *gcc*

Note that micro-TLB performance for *tree* is poor. With 4K byte pages, a two entry micro-TLB absorbs about 0.10 CPI. Micro-TLB performance for *tree* begins to improve rapidly beyond four micro-TLB entries. There are two effects that could conceivably contribute to the degraded performance: the working set size of the computation, and conflicts between the garbage collector and the rest of the computation. The garbage collector and the program behave as independent co-routines or threads in the single address space. In the case of *tree*, most of the observed miss rate is due to locality properties of the compiled Scheme code, as co-routine exchanges between the garbage collector and the program execution are much too infrequent[16] to account for the observed miss rates.

Although multi-thread effects were not important with *tree*, it is worth noting that in as much as threads tend to execute in independent code localities, tightly coupled threads executing in a single address space will lead to degraded micro-TLB performance. This effect will be exaggerated with a very small micro-TLB. The importance of this effect on a uniprocessor depends on usage patterns for threads. If tightly coupled threads were a common programming paradigm, these effects could potentially have an impact.

However, it is not clear that fine grain threads on a uniprocessor as a paradigm merits special support, as coarse grain threads may dominate actual usage. The UNIX server in Mach 3.0 is another example from our workloads of a threaded address space. Its multi-threaded structure has little impact on TLB behavior, as the threads are relatively heavy-weight. Furthermore, fine grained threads introduce other performance problems such as cache interference and processor state management overhead, which tend to make micro-TLB performance irrelevant. Overall, threaded programs are not common enough for a meaningful analysis of their impact on micro-TLB performance to be made at this time.

7.2.1.1. The Impact of System Activity

For our model of system activity, we assume all Ultrix instruction references and all Mach 3.0 kernel instruction references are unmapped and isolated from the micro-TLB. The result is that for system intensive programs, system activity has little or no impact on the total number of micro-TLB misses, and because of the increase in the instruction reference count, the overall miss rate (ratio of misses per instruction) is significantly lower. Tables 7-2 and 7-3 show micro-TLB miss rates for Mach 3.0 and Ultrix, respectively.

Tables 7-2 and 7-3 show that due to the addition of unmapped references, system effects induce improved micro-TLB performance for both systems and across the entire range of workloads. The behavior of Mach 3.0 is uniformly worse than Ultrix due to the mapped system references from the UNIX server and the emulation library. Behavior for Mach is still consistently better than for experiments which exclude system activity.

[16]The *tree* execution takes about 147 seconds, of which 2 seconds are spent collecting garbage. Garbage collections were observed to occur every 8-15 seconds.

	1x4KB	2x4KB	4x4KB	8x4KB	1x16KB	2x16KB	4x16KB	8x16KB
sed	0.030069	0.006671	0.000546	0.000193	0.010200	0.000752	0.000265	0.000045
egrep	0.001886	0.000166	0.000064	0.000022	0.000412	0.000086	0.000031	0.000006
yacc	0.011419	0.002265	0.000499	0.000031	0.004603	0.000439	0.000036	0.000006
gcc	0.015848	0.007420	0.003581	0.001496	0.010293	0.003670	0.001101	0.000122
compress	0.003199	0.001404	0.000269	0.000095	0.001638	0.000341	0.000122	0.000018
ab	0.016548	0.006459	0.003045	0.001196	0.011509	0.003328	0.001190	0.000215
espresso	0.007852	0.002447	0.000706	0.000026	0.006937	0.001559	0.000113	0.000002
lisp	0.025192	0.010563	0.003484	0.000429	0.019772	0.007281	0.000092	0.000001
eqntott	0.005723	0.000713	0.000010	0.000003	0.005337	0.000484	0.000005	0.000001
fpppp	0.001055	0.000391	0.000170	0.000019	0.000637	0.000139	0.000047	0.000001
doduc	0.010767	0.002669	0.001230	0.000137	0.004752	0.001820	0.000352	0.000002
liv	0.009380	0.000164	0.000071	0.000026	0.000418	0.000092	0.000035	0.000006
tomcatv	0.000056	0.000024	0.000012	0.000003	0.000045	0.000012	0.000004	0.000000

Table 7-2: Micro-TLB Miss Rates for Mach 3.0

This table shows miss rates for a range FIFO micro-TLB configurations. System references were included for these experiments.

	1x4KB	2x4KB	4x4KB	8x4KB	1x16KB	2x16KB	4x16KB	8x16KB
sed	0.029458	0.006154	0.000001	0.000000	0.007520	0.000000	0.000000	0.000000
egrep	0.001435	0.000009	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
yacc	0.010906	0.002064	0.000415	0.000000	0.004059	0.000349	0.000000	0.000000
gcc	0.013454	0.006867	0.003479	0.001551	0.007638	0.003299	0.001084	0.000123
compress	0.001387	0.000830	0.000000	0.000000	0.000001	0.000000	0.000000	0.000000
ab	0.013578	0.005451	0.002637	0.001019	0.008320	0.002761	0.000853	0.000162
espresso	0.007661	0.002391	0.000682	0.000018	0.006773	0.001546	0.000106	0.000000
lisp	0.025343	0.010618	0.003466	0.000421	0.019836	0.007281	0.000000	0.000000
eqntott	0.005667	0.000691	0.000000	0.000000	0.005288	0.000481	0.000000	0.000000
fpppp	0.000910	0.000337	0.000148	0.000013	0.000508	0.000111	0.000028	0.000000
doduc	0.010653	0.002616	0.001211	0.000128	0.004620	0.001793	0.000338	0.000000
liv	0.008958	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
tomcatv	0.000018	0.000007	0.000004	0.000001	0.000013	0.000003	0.000000	0.000000

Table 7-3: Micro-TLB Miss Rates for DEC Ultrix

This table shows miss rates for a range FIFO micro-TLB configurations. System references were included for these experiments.

7.2.2. Instruction TLBs

We simulated instruction TLBs with sizes in powers of two from 8 to 64 entries and page sizes in powers two from 4K to 64K bytes. In examples used to illustrate TLB performance, an approximation of 100 cycles is used for the TLB miss penalty. This figure is based on a model where a TLB miss requires two memory references corresponding to a page table lookup, and where the latency of these memory references dominates the TLB miss penalty. The penalty of 100 cycles corresponds to a futuristic microprocessor with a cycle time of under five nanoseconds. It is meant to be somewhat less than the time for two references to main memory and somewhat more than the time for two references to an off-chip cache. Under these assumptions, with TLB performance of 10000 instructions per miss, the contribution of the TLB to CPI is 0.01. This is our (somewhat arbitrary) lower bound on reasonable TLB performance. Again we first discuss measurements of user activity only, then extend for system behavior.

The behavior of single-benchmark workloads in instruction-only TLBs is mostly uninteresting, as most of the sample workloads have small text, well below the capacities of the TLBs we tested. *Gcc*, with 688 K bytes of text, is one of the few SPEC workloads that presents a significant demand on resources. Figure 7-6 illustrates the behavior for *gcc*. Solid lines connect TLB configurations with the same page size. Dashed lines connect TLBs that map the same amount of memory. The amount of memory mapped by a TLB will be referred to as its *mapping size*, to discriminate between that measure of size and others, such as the number of lines in a TLB.

As with the micro-TLB, there are three regimes of behavior to be observed. The placement of data points is more compact in the lower portion of the graph, with relatively little improvement for larger TLB configurations. This represents thrashing, where TLB resources are well below the working set size of the program. In the next region, performance improves quickly as working set size is approached. This figure shows that *gcc* approaches the TLB resources to map its working set with an instruction TLB mapping size of 512K bytes. Once the working set size of the program has been reached, increasing the mapping size has a reduced effect on performance, and the points again become more closely spaced. Such behavior occurs at the top of the graph. These three regimes of behavior become more pronounced for shared and data TLBs, as in Figure 7-8.

In the lower part of the graph, the dashed lines that connect TLBs with the same mapping size tend to slope upward slightly, while at the top of the graph they slope down. An application that uses sparse, non-contiguous data tends to have better TLB performance with more smaller pages of memory than with a few larger pages. Such behavior occurs in the lower part of the graph. In the upper part of the graph, the dashed lines tend to slope down, hence better performance with fewer larger pages. As a TLB becomes large enough to map all of a program's working set, smaller pages mean that TLB misses occur for each of several small pages, rather than once for a single large page. Similar behavior occurs when a program accesses contiguous data, as in Figure 7-8.

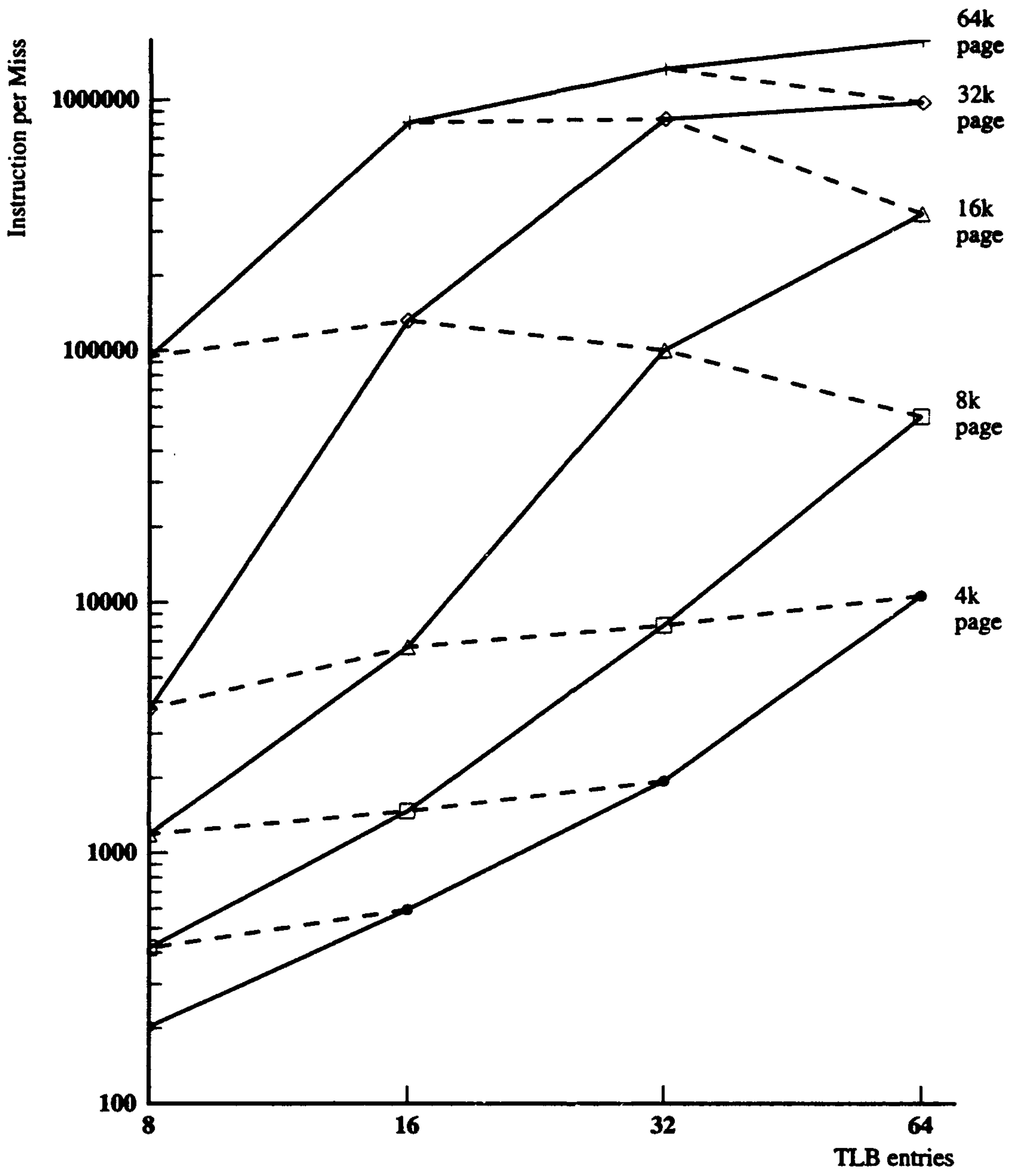

Figure 7-6: *Gcc* Instruction TLB Behavior

This figure illustrates instruction TLB behavior for the *gcc* workload. The TLBs used Random Replacement and full associativity. System activity is not included. As in Figure 7-1, three domains of behavior can be identified, corresponding configurations that thrash, approach working set size, and configurations where working set size has been reached.

Figure 7-7 illustrates instruction TLB performance for the multi-task mix. For this workload, the flat dashed lines suggest that mapping size is entirely responsible for determining TLB performance, and that the configuration of page size and number of entries has little effect. TLB performance crosses the 10000 instruction per miss performance

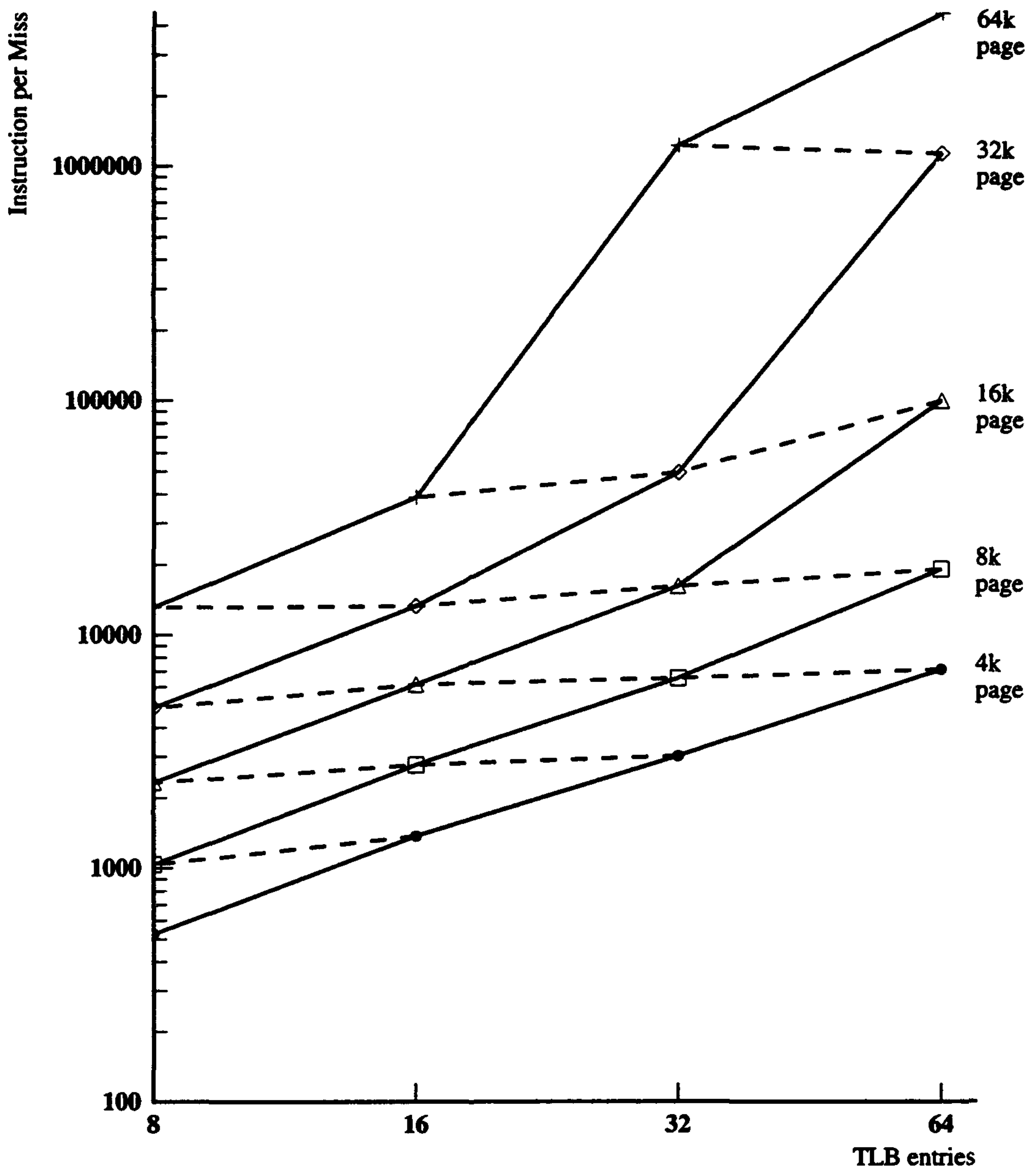

Figure 7-7: Multi-task instruction TLB behavior

This figure illustrates instruction TLB behavior for the mixA multitasking workload. System activity is not included. All TLBs are fully associative with random replacement.

boundary at a mapping size of 512K bytes, and appears to have reached the working set size for mapping sizes over 2 megabytes.

Table 7-4 gives miss rates for selected benchmarks for a number of instruction-only TLB configurations. The benchmarks not included have very low miss rates.

	16x4KB	32x4KB	64x4KB	16x16KB	32x16KB
gcc	**0.001688**	**0.000516**	0.000094	0.000151	0.000010
magic	**0.000275**	0.000059	0.000000	0.000041	0.000000
tree	**0.000309**	0.000000	0.000000	0.000000	0.000000
ccom	**0.001969**	0.000032	0.000003	0.000001	0.000001
mixA	**0.000728**	**0.000331**	**0.000140**	**0.000163**	0.000062

Table 7-4: Instruction TLB Miss Rates

This table shows instruction TLB behavior for the five workloads. Other workloads had very low miss rates. These experiments use user references only. All TLBs are fully associative with random replacement.

7.2.2.1. The Impact of System Activity

With our model of system memory, kernel activity is largely isolated from TLB behavior. So, as with the micro-TLB, system activity has little impact on the number of instruction TLB misses, and the increase in instruction count makes the miss rate lower. Tables 7-5 and 7-6 show instruction TLB miss rates for Mach 3.0 and Ultrix.

	8x4KB	16x4KB	32x4KB	8x8KB	16x8KB	32x8KB	8x16KB	16x16KB	32x16KB
sed	0.000945	0.000500	0.000225	0.000596	0.000281	0.000087	0.000360	0.000123	0.000022
egrep	0.000153	0.000078	0.000031	0.000101	0.000044	0.000012	0.000058	0.000020	0.000003
yacc	0.000234	0.000127	0.000060	0.000149	0.000075	0.000021	0.000089	0.000032	0.000003
gcc	0.003713	0.001394	0.000485	0.001861	0.000610	0.000154	0.000785	0.000190	0.000033
compress	0.000509	0.000279	0.000124	0.000330	0.000162	0.000036	0.000197	0.000068	0.000008
ab	0.003566	0.001681	0.000635	0.002144	0.000816	0.000265	0.001065	0.000354	0.000058
espresso	0.000127	0.000039	0.000013	0.000059	0.000020	0.000005	0.000029	0.000008	0.000001
lisp	0.001397	0.000022	0.000001	0.000303	0.000009	0.000000	0.000013	0.000002	0.000000
eqntott	0.000016	0.000003	0.000001	0.000010	0.000002	0.000000	0.000007	0.000001	0.000000
fpppp	0.000687	0.000196	0.000010	0.000186	0.000017	0.000002	0.000022	0.000004	0.000000
doduc	0.000648	0.000181	0.000029	0.000219	0.000057	0.000002	0.000079	0.000013	0.000000
liv	0.000145	0.000064	0.000026	0.000085	0.000035	0.000012	0.000045	0.000014	0.000004
tomcatv	0.000015	0.000003	0.000000	0.000009	0.000001	0.000000	0.000005	0.000000	0.000000

Table 7-5: Instruction TLB Miss Rates for Mach 3.0

This table shows miss rates for a instruction TLB configurations. All TLBs are fully associative and use random replacement. System references were included for these experiments. Idle loop activity is not included.

Tables 7-5 and 7-6 show that instruction TLB miss rates are worse for Mach 3.0 than for Ultrix, due to the additional mapped instruction references from the Mach UNIX server and emulation library. Considering micro-TLB and instruction TLB experiments together, miss rates for Mach 3.0 and Ultrix are comparable during thrashing or after working set size has been reached, but it often takes a larger TLB for Mach 3.0 to get out of the thrashing region.

	8x4KB	16x4KB	32x4KB	8x8KB	16x8KB	32x8KB	8x16KB	16x16KB	32x16KB
sed	0.000002	0.000002	0.000002	0.000002	0.000002	0.000002	0.000002	0.000002	0.000002
egrep	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
yacc	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
gcc	0.001014	0.000509	0.000196	0.000853	0.000315	0.000065	0.000589	0.000118	0.000009
compress	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ab	0.003117	0.001347	0.000497	0.001706	0.000575	0.000212	0.000732	0.000248	0.000026
espresso	0.000013	0.000001	0.000000	0.000007	0.000000	0.000000	0.000001	0.000000	0.000000
lisp	0.001171	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
eqntott	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
fpppp	0.000008	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
doduc	0.000083	0.000000	0.000000	0.000064	0.000000	0.000000	0.000032	0.000000	0.000000
liv	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
tomcatv	0.000002	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Table 7-6: Instruction TLB Miss Rates for DEC Ultrix

This table shows miss rates for a instruction TLB configurations. All TLBs are fully associative and use random replacement. System references were included for these experiments. Idle loop activity is not included.

7.3. Data and Shared TLB Results

Data-only TLBs and shared TLBs behave similarly. As most programs have well behaved instruction reference patterns, behavior in the shared case is dominated by data references. Although the following discussion is mostly in terms of shared TLBs, the analysis can be applied to the data-only case as well.

We simulated shared TLBs with sizes in powers of two from 32 to 256 entries, page sizes in powers of two from 4K to 64K bytes, and with LRU and Random replacement policies. All TLBs simulated were fully associative.

Figure 7-8 shows the shared TLB behavior for *mat300*, one of the worst behaved workloads. Note that the TLBs in this figure are a factor of four larger than those considered with instruction TLBs, ranging from 32 to 256 entries as opposed to 8 to 64 entries. With a miss penalty of 100 cycles and a 64 entry TLB with 4K byte pages, *mat300* spends 5 CPI on the TLB. *Mat300* does matrix operations on three matrices with a total size of approximately 2.5 megabytes. The contents of these three matrices are accessed in regular patterns, sometimes sequentially and sometimes stepping by columns. This explains the poor behavior when the TLB maps less than 2.5 megabytes, and rapid improvement as that barrier is reached and surpassed. Observe that lines connecting TLBs with the same mapping size always slope down, consistent with the observation that *mat300* accesses its data sequentially.

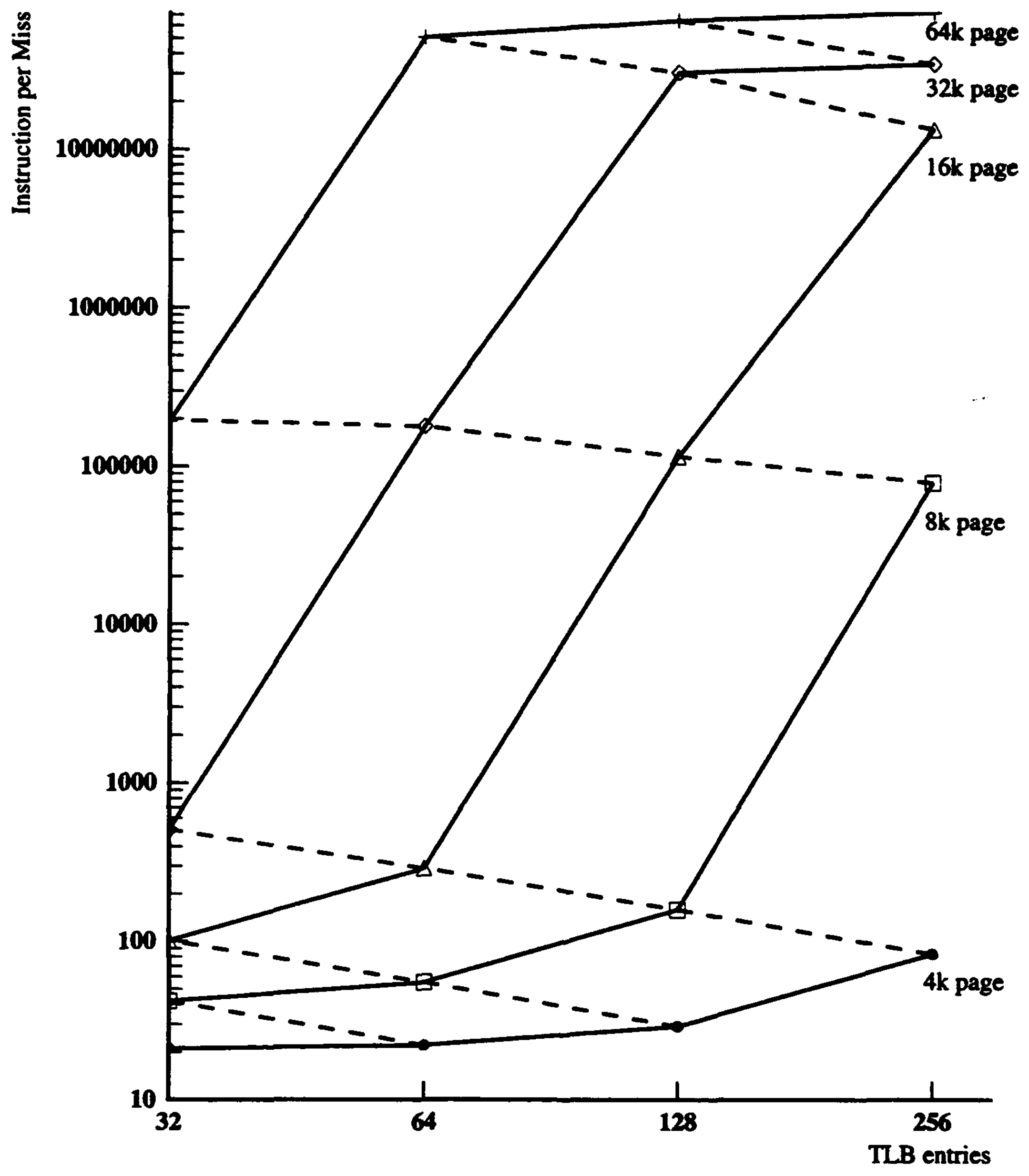

Figure 7-8: *Mat300* Shared TLB Behavior

This figure illustrates shared TLB behavior for the *mat300* workload. Miss behavior for *mat300* is dominated by data references. All TLBs are fully associative with random replacement. System activity is not included.

Figure 7-9 shows the TLB behavior for *tree* running with a 10 megabyte heap. Memory is allocated from 5 megabytes of the heap, while the other 5 megabytes is reserved for garbage collection. As expected, performance improves steadily through a mapping size of 5 megabytes (16k pages × 256 entries), after which the rate of improvement begins to diminish, shown by the downward sloping lines connecting configurations

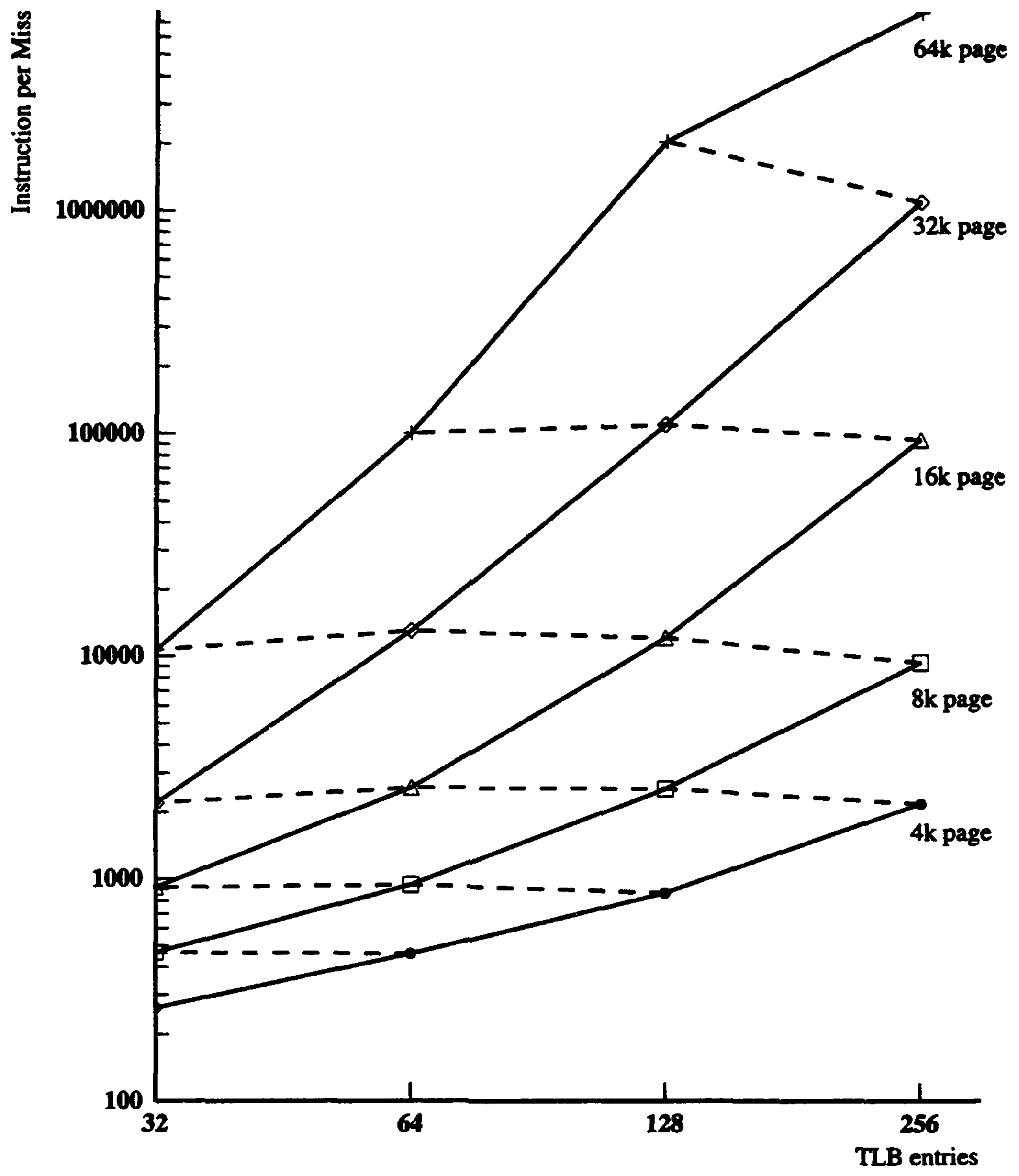

Figure 7-9: *Tree* Shared TLB Behavior

This figure illustrates shared TLB behavior for the *tree* workload. All TLBs are fully associative with random replacement. System activity is not included.

with equal mapping sizes. Note that below the 5 megabyte boundary, the lines connecting TLBs with the same mapping size are nearly horizontal. This indicates that TLB performance for this benchmark depends exclusively on the mapping size. Other TLB parameters are unimportant.

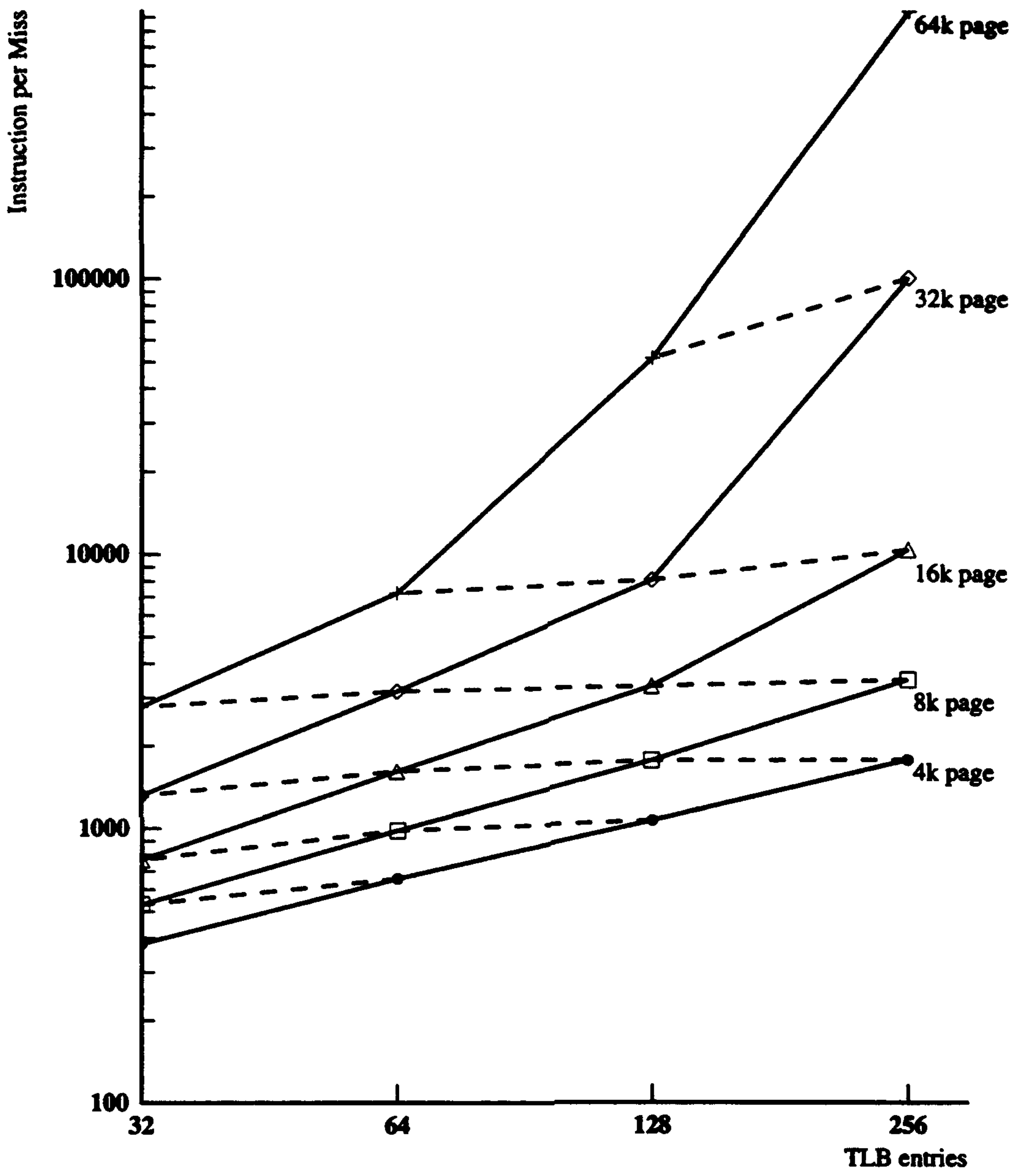

Figure 7-10: Multiprocess Mix TLB Behavior

This figure illustrates shared TLB behavior for the multitasking workload. All TLBs are fully associative with random replacement. System activity is not included.

As a last illustration of shared TLB behavior, Figure 7-10 shows a plot for the multitask mix. Several conclusions are immediate. First, as for *tree*, the mapping size of the TLBs is the dominant factor in performance. Only after a mapping size of eight megabytes do the dashed lines stop looking horizontal. Also, most of the plot is fairly compact. Data points become less compact beyond a mapping size of eight megabytes as the working set size is approached. Lastly, for all TLBs with a mapping size of one megabyte or less, assuming a 100 cycle miss penalty, at least 0.05 CPI is lost to the TLB — a significant performance penalty. This suggests that if a machine with a TLB is to execute such a workload efficiently, the TLB must have a significantly larger mapping size.

	64x4KB	128x4KB	256x4KB	64x16KB	64x32KB
doduc	0.000014	0.000000	0.000000	0.000000	0.000000
eqntott	**0.000410**	**0.000162**	0.000002	0.000020	0.000000
espress	0.000004	0.000001	0.000001	0.000000	0.000000
fpppp	0.000001	0.000000	0.000000	0.000000	0.000000
gcc	**0.001143**	**0.000244**	**0.000132**	0.000053	0.000013
lisp	0.000000	0.000000	0.000000	0.000000	0.000000
mat300	**0.049097**	**0.036378**	**0.012669**	**0.003461**	0.000006
nasa7	**0.016526**	**0.011955**	**0.005183**	**0.002024**	0.000000
spice	0.000052	0.000005	0.000000	0.000000	0.000000
tomcatv	**0.000151**	**0.000138**	0.000121	0.000033	0.000012
ccom	**0.000283**	0.000009	0.000006	0.000005	0.000002
sed	0.000017	0.000017	0.000015	0.000006	0.000005
tree	**0.004683**	**0.002749**	**0.001015**	**0.000224**	0.000069
magic	**0.000644**	**0.000139**	0.000003	**0.000106**	0.000086
mixA	**0.001527**	**0.000937**	**0.000563**	**0.000620**	**0.000316**

Table 7-7: Shared TLB Miss Rates.

This table shows miss rates for shared TLBs with user activity only. All TLBs use Random Replacement and are, fully associative. Experiments for this table used user references only.

Table 7-7 shows TLB miss rates for the ten SPECmarks and several other workloads of interest. This table is highlighted to indicate where the TLB penalty is more than 0.01 CPI. With the exceptions of *nasa7* and *mat300*, both of which are oriented towards scientific/vector machines, the SPECmarks perform reach this performance limit with a 64x16k TLB, suggesting that such a configuration provides adequate TLB performance. The smaller configurations don't perform as well. This suggests that sometime in the near future, TLBs with larger mapping sizes are needed, especially for machines running numeric programs.

Table 7-8 gives miss rates for a number of data-only TLB configurations.

Figure 7-11 compares miss rates for shared and split TLBs. All TLBs use 64 entries and 4K byte pages. For all but *gcc* and the multi-task mix, the instruction miss rates are inconsequential. Note that the sum of the split instruction and data miss rates is generally less than the miss rate for the shared TLB. This difference represents competition for TLB entries between instruction and data references. This comparison is not meant to

	64x4KB	128x4KB	256x4KB	64x16KB	128x16KB
doduc	0.000000	0.000000	0.000000	0.000000	0.000000
eqntott	**0.000355**	**0.000140**	0.000002	0.000016	0.000000
espress	0.000001	0.000001	0.000000	0.000000	0.000000
fpppp	0.000000	0.000000	0.000000	0.000000	0.000000
gcc	**0.000152**	**0.000104**	0.000067	0.000022	0.000014
lisp	0.000000	0.000000	0.000000	0.000000	0.000000
mat300	**0.045666**	**0.034275**	**0.012050**	**0.003460**	0.000008
nasa7	**0.016521**	**0.011947**	**0.005178**	**0.002022**	0.000001
spice	0.000034	0.000000	0.000000	0.000000	0.000000
tomcatv	**0.000149**	**0.000137**	0.000120	0.000033	0.000021
ccom	0.000005	0.000004	0.000003	0.000001	0.000001
sed	0.000011	0.000010	0.000009	0.000005	0.000004
tree	**0.001804**	**0.001002**	**0.000420**	**0.000288**	0.000076
magic	**0.000270**	0.000038	0.000003	0.000059	0.000002
mixA	**0.001358**	**0.000866**	**0.000511**	**0.000546**	**0.000258**

Table 7-8: Data TLB Miss Rates.

This table shows miss rates for data-only TLBs, with user activity only. The simulated TLBs use random replacement and are fully associative. Experiments for this table used user references only.

suggest two specific implementation alternatives, as the split TLBs illustrated use twice the resources of the shared TLB.

7.3.1. The Impact of System Activity

In our model of system activity, following the example of the DECstation system architecture, three types of kernel data structures are accessed through mapped memory:

- The buffer cache of disk data
- Page table information
- Per-process data (the user structure).

The impact of system activity on data and shared TLB behavior, with respect to user-only behavior, depends on three considerations:

- Kernel contribution to instruction count
- Access patterns for mapped kernel data structures
- Contributions from user-level system contexts (Mach 3.0).

The impact of system activity on the behavior of data and shared TLBs is dependent on workload and determined by the level of activity and interactions, as per the above three considerations. Tables 7-9 and 7-10 show Mach 3.0 and Ultrix miss rates for data TLBs. Tables 7-11 and 7-12 show Mach 3.0 and Ultrix miss rates for shared TLBs.

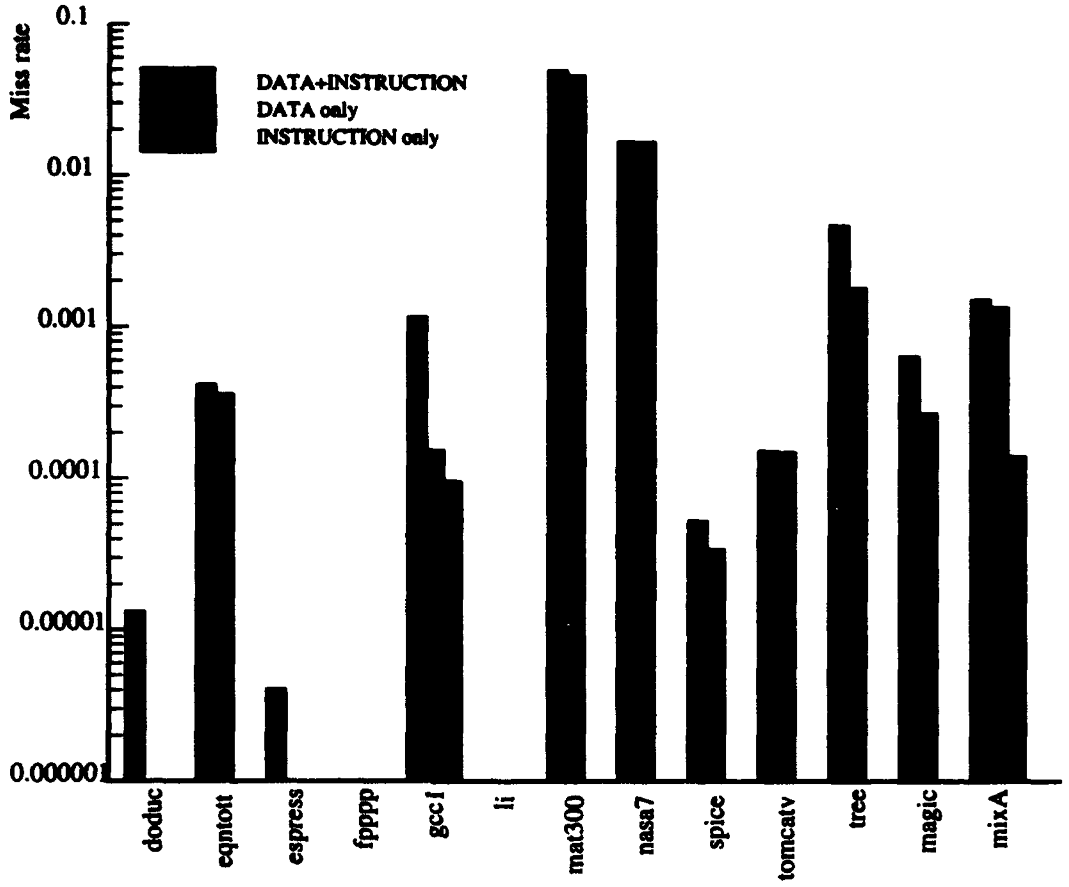

Figure 7-11: Shared vs. Split TLBs

This figure compares miss rates for shared and split TLBs, with 64 entries and 4K byte pages. All TLBs were fully associative with random replacement. System activity is not included.

	16x4KB	32x4KB	64x4KB	16x8KB	32x8KB	64x8KB	16x16KB	32x16KB	64x16KB
sed	0.001220	0.000679	0.000306	0.000985	0.000515	0.000180	0.000730	0.000314	0.000086
egrep	0.000232	0.000132	0.000058	0.000189	0.000102	0.000036	0.000140	0.000062	0.000019
yacc	0.000620	0.000223	0.000106	0.000332	0.000175	0.000069	0.000243	0.000116	0.000030
gcc	0.004011	0.001022	0.000335	0.002053	0.000624	0.000159	0.001016	0.000346	0.000052
compress	0.015519	0.008436	0.003479	0.009769	0.004073	0.000318	0.005235	0.000352	0.000106
ab	0.004742	0.001813	0.000721	0.003034	0.001224	0.000451	0.001782	0.000718	0.000218
espresso	0.000586	0.000078	0.000027	0.000118	0.000048	0.000015	0.000073	0.000028	0.000006
lisp	0.000331	0.000015	0.000000	0.000030	0.000004	0.000000	0.000013	0.000001	0.000000
eqntott	0.001569	0.000738	0.000415	0.001051	0.000409	0.000171	0.000554	0.000176	0.000032
fpppp	0.001657	0.000078	0.000008	0.000238	0.000025	0.000005	0.000041	0.000009	0.000002
doduc	0.001626	0.000116	0.000011	0.000227	0.000044	0.000006	0.000075	0.000016	0.000002
liv	0.000205	0.000111	0.000056	0.000161	0.000083	0.000037	0.000116	0.000053	0.000019
tomcatv	0.000263	0.000184	0.000158	0.000132	0.000095	0.000080	0.000070	0.000050	0.000039

Table 7-9: Data Miss Rates for Mach 3.0

This table shows miss rates for a range of data TLB configurations. All TLBs are fully associative and use random replacement. System references were included for these experiments. Idle-loop activity was excluded.

	16x4KB	32x4KB	64x4KB	16x8KB	32x8KB	64x8KB	16x16KB	32x16KB	64x16KB
sed	0.000161	0.000076	0.000035	0.000086	0.000040	0.000018	0.000059	0.000021	0.000012
egrep	0.000023	0.000012	0.000007	0.000013	0.000006	0.000003	0.000009	0.000004	0.000002
yacc	0.000323	0.000026	0.000010	0.000037	0.000009	0.000005	0.000015	0.000006	0.000003
gcc	0.003627	0.000430	0.000086	0.001342	0.000105	0.000024	0.000310	0.000025	0.000010
compress	0.017285	0.009256	0.003662	0.010569	0.004255	0.000047	0.005510	0.000060	0.000013
ab	0.003344	0.000638	0.000131	0.001514	0.000219	0.000045	0.000406	0.000059	0.000022
espresso	0.000506	0.000013	0.000002	0.000028	0.000002	0.000001	0.000004	0.000001	0.000000
lisp	0.000149	0.000000	0.000000	0.000002	0.000000	0.000000	0.000000	0.000000	0.000000
eqntott	0.001591	0.000746	0.000410	0.001047	0.000399	0.000155	0.000552	0.000162	0.000015
fpppp	0.001663	0.000036	0.000001	0.000222	0.000000	0.000000	0.000002	0.000000	0.000000
doduc	0.001595	0.000057	0.000000	0.000144	0.000000	0.000000	0.000001	0.000000	0.000000
liv	0.000017	0.000010	0.000007	0.000009	0.000006	0.000004	0.000007	0.000004	0.000003
tomcatv	0.000273	0.000180	0.000153	0.000121	0.000086	0.000073	0.000059	0.000042	0.000034

Table 7-10: Data Miss Rates for DEC Ultrix

This table shows miss rates for a range data TLB configurations. All TLBs are fully associative and use random replacement. System references were included for these experiments. Idle-loop activity was excluded.

	16x4KB	32x4KB	64x4KB	16x8KB	32x8KB	64x8KB	16x16KB	32x16KB	64x16KB
sed	0.003119	0.001677	0.000850	0.002380	0.001209	0.000541	0.001728	0.000793	0.000246
egrep	0.000551	0.000298	0.000149	0.000426	0.000221	0.000093	0.000312	0.000144	0.000043
yacc	0.001933	0.000535	0.000273	0.001018	0.000375	0.000178	0.000518	0.000243	0.000083
gcc	0.015698	0.004646	0.001339	0.009441	0.002262	0.000613	0.004521	0.000941	0.000228
compress	0.019367	0.009492	0.003980	0.011993	0.004817	0.000558	0.006760	0.000640	0.000234
ab	0.013206	0.005970	0.002238	0.009309	0.003472	0.001257	0.005291	0.001924	0.000546
espresso	0.002306	0.000210	0.000072	0.000516	0.000114	0.000039	0.000178	0.000067	0.000018
lisp	0.006013	0.000305	0.000014	0.000856	0.000037	0.000002	0.000098	0.000016	0.000000
eqntott	0.002297	0.000858	0.000458	0.001604	0.000488	0.000200	0.000848	0.000222	0.000053
fpppp	0.004485	0.000921	0.000070	0.002058	0.000115	0.000021	0.000194	0.000043	0.000007
doduc	0.007365	0.000669	0.000094	0.002576	0.000194	0.000024	0.000286	0.000071	0.000007
liv	0.000573	0.000276	0.000134	0.000424	0.000194	0.000088	0.000303	0.000126	0.000045
tomcatv	0.000329	0.000205	0.000170	0.000171	0.000109	0.000088	0.000096	0.000060	0.000045

Table 7-11: Shared Miss Rates for Mach 3.0

This table shows miss rates for a range shared instruction and data TLB configurations. All TLBs are fully associative and use random replacement. System references were included for these experiments. Idle-loop activity was excluded.

	16x4KB	32x4KB	64x4KB	16x8KB	32x8KB	64x8KB	16x16KB	32x16KB	64x16KB
sed	0.000233	0.000096	0.000055	0.000110	0.000049	0.000032	0.000079	0.000035	0.000021
egrep	0.000031	0.000015	0.000009	0.000015	0.000008	0.000005	0.000010	0.000005	0.000003
yacc	0.001204	0.000061	0.000013	0.000383	0.000015	0.000006	0.000026	0.000006	0.000004
gcc	0.016511	0.004091	0.000788	0.009318	0.001458	0.000166	0.003821	0.000278	0.000032
compress	0.020510	0.009866	0.003848	0.012336	0.004677	0.000054	0.006655	0.000067	0.000017
ab	0.011311	0.004221	0.001092	0.007213	0.001723	0.000411	0.003072	0.000591	0.000058
espresso	0.002152	0.000082	0.000006	0.000349	0.000009	0.000002	0.000037	0.000002	0.000000
lisp	0.005628	0.000014	0.000000	0.000542	0.000000	0.000000	0.000002	0.000000	0.000000
eqntott	0.002285	0.000841	0.000437	0.001573	0.000460	0.000170	0.000823	0.000193	0.000020
fpppp	0.004364	0.000935	0.000006	0.001956	0.000023	0.000000	0.000105	0.000001	0.000000
doduc	0.007255	0.000574	0.000004	0.002444	0.000065	0.000000	0.000152	0.000000	0.000000
liv	0.000062	0.000010	0.000008	0.000010	0.000006	0.000004	0.000007	0.000005	0.000003
tomcatv	0.000317	0.000189	0.000157	0.000139	0.000090	0.000075	0.000067	0.000044	0.000035

Table 7-12: Shared TLB Miss Rates for DEC Ultrix

This table shows miss rates for a range shared instruction and data TLB configurations. All TLBs are fully associative and use random replacement. System references were included for these experiments. Idle-loop activity was excluded.

The workload *sed* is an example where the addition of system activity induces worse TLB miss ratios. This is due in part to a relatively high ratio of buffer cache activity to computation. The effect is exaggerated as *sed* is run three times, with the *sed* binary and input files retrieved twice from the buffer cache. For *gcc*, system activity causes improved TLB miss ratios. In this case, the ratio of buffer cache activity to computation is lower, with the *gcc* binary loaded once directly from disk. Additional system instructions make the ratio of misses to instructions lower. Lastly, workloads such as *tomcatv* require little system activity so the impact of this activity is slight, contributing few TLB misses and few system instructions.

In our comparisons of Mach 3.0 and Ultrix TLB behavior, we have seen consistently higher miss rates for Mach 3.0 than for Ultrix, due to the mapped system contexts in Mach 3.0. Although this general rule also applies to data-only and shared TLBs, the workloads *compress* and *eqntott* are interesting exceptions. For these two workloads, Ultrix and Mach 3.0 get comparable numbers of data-only TLB references and misses. However, higher instruction counts for Mach 3.0 makes the miss ratio lower for Mach 3.0. Such behavior also occurs for *compress* for smaller sizes of shared TLBs.

7.4. Variable Size TLB Entries

An interesting question for future work is how to make use of the variable size TLB entries that have appeared in recent architectures [33, 56]. Maps of the dynamic patterns of memory access are useful to understand this problem. Figure 7-12 shows the pattern of data memory accesses for *mat300*. Page address varies in the *x* dimension, from 0x10000000 on the left to 0x1021e000 on the right; a range of about 2.2 megabytes. Instruction count (i.e., time) varies along the *y* dimension, ranging from 0 at the top to 2.77 billion at the bottom. Each dark square represents one or more accesses to a 16K byte page during an interval of 1 million instructions.

The three matrices used by *mat300* are clearly visible from the usage patterns in the address space. The compactness and predictability of the *mat300* accesses show that the use of larger pages could virtually eliminate TLB misses, provided that adequate cache and memory resources were available.

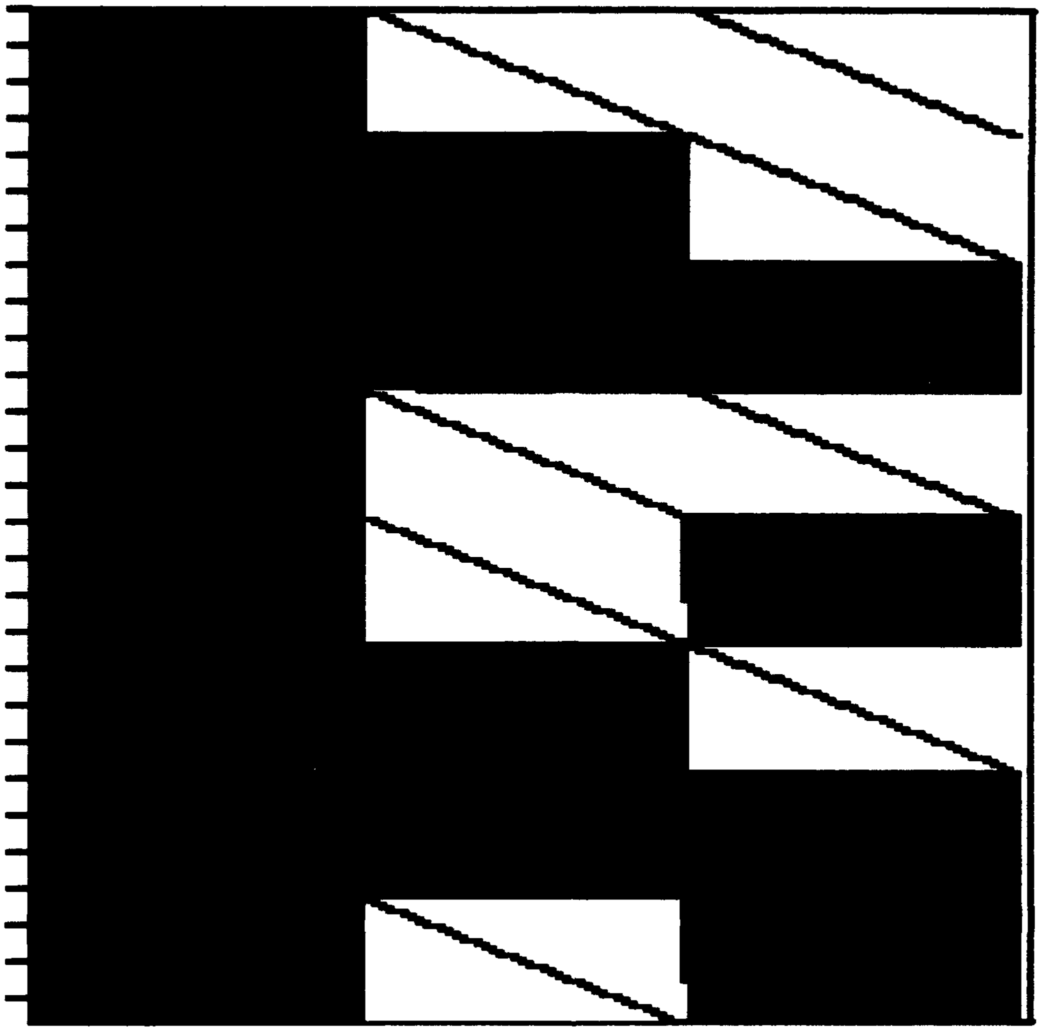

Figure 7-12: *mat300* Data Memory Access Patterns

This figure is a graphical representation of user data access patterns for *mat300*. The x axis represents data addresses, ranging from `0x10000000` on the left to `0x1021e000` on the right. The y axis represents time in terms of instructions, with instruction 0 at the top and instruction 2770000000 at the bottom of the figure. Each dark point in the figure represents one or more references to a memory page during a 10 million instruction interval. This figure shows that the memory access patterns of *mat300* are regular and predictable, such that large TLB pages could easily be applied.

Tree, the lisp benchmark, also shows interesting data reference patterns, illustrated in Figure 7-13. Note that a page size of 64K bytes was used. The address space represented in this figure is about 11 megabytes. The descending staircase pattern shows the behavior of the memory allocator as it walks across the heap. Solid vertical bands show where garbage collection has compacted the heap into frequently accessed regions. The pattern of memory references for *tree* is sparse relative to *mat300*. This behavior, along with the size of the address space, suggests that lisp workloads such as *tree* are relatively poor candidates for variable size pages.

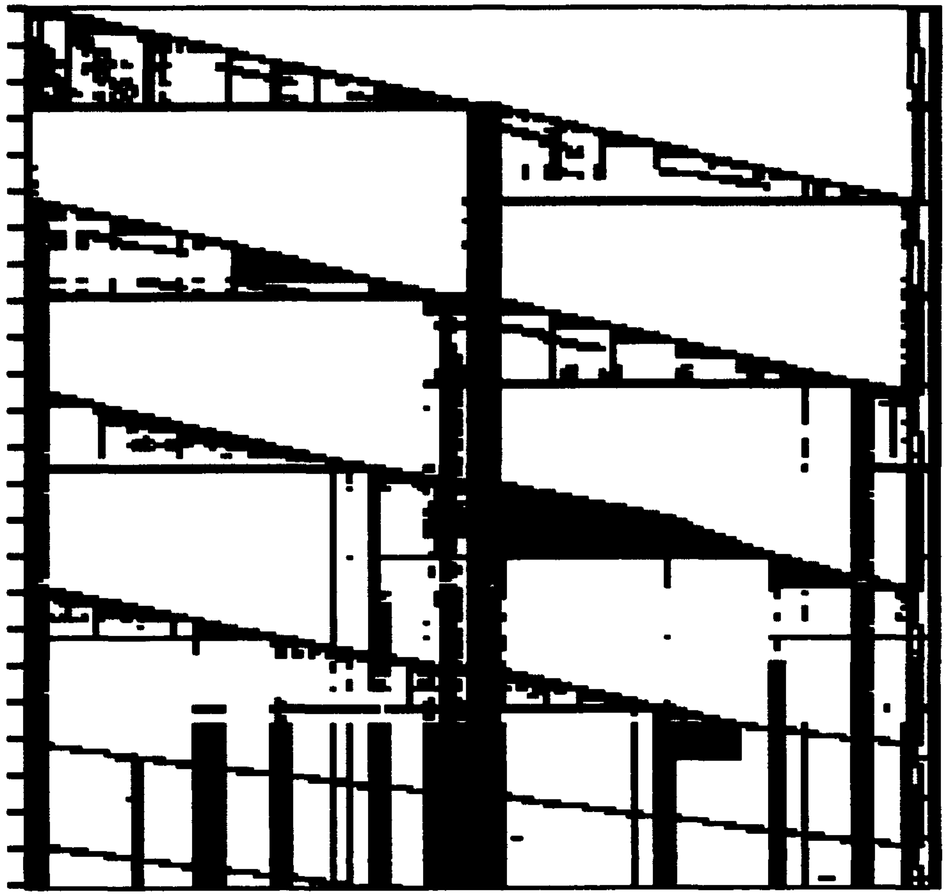

Figure 7-13: *tree* Data Memory Access Patterns

This figure is a graphical representation of user data access patterns for *tree*. The *x* axis represents data addresses, ranging from `0x10000000` on the left to `0x10a5c000` on the right. The *y* axis represents time in terms of instructions, with instruction 0 at the top and instruction 2410000000 at the bottom of the figure. Each dark point in the figure represents one or more references to a memory page during a 10 million instruction interval. This figure shows that memory access patterns for *tree* are spread sparsely across the data segment, making it difficult to apply large TLB pages without substantial memory fragmentation.

Interesting patterns of reference are the exception rather than the rule in memory access patterns. Most of the benchmarks concentrate on a small number of unclustered pages, resulting in a few dark vertical bars from the top to the bottom of the map with occasional horizontal excursions.

Figure 7-14 shows a map of instruction references for *gcc*. Each point represents one or more references to a 4k byte page during an interval of 100000 instructions. The address space spanned in this figure is 684K bytes, the largest text segment of any of the

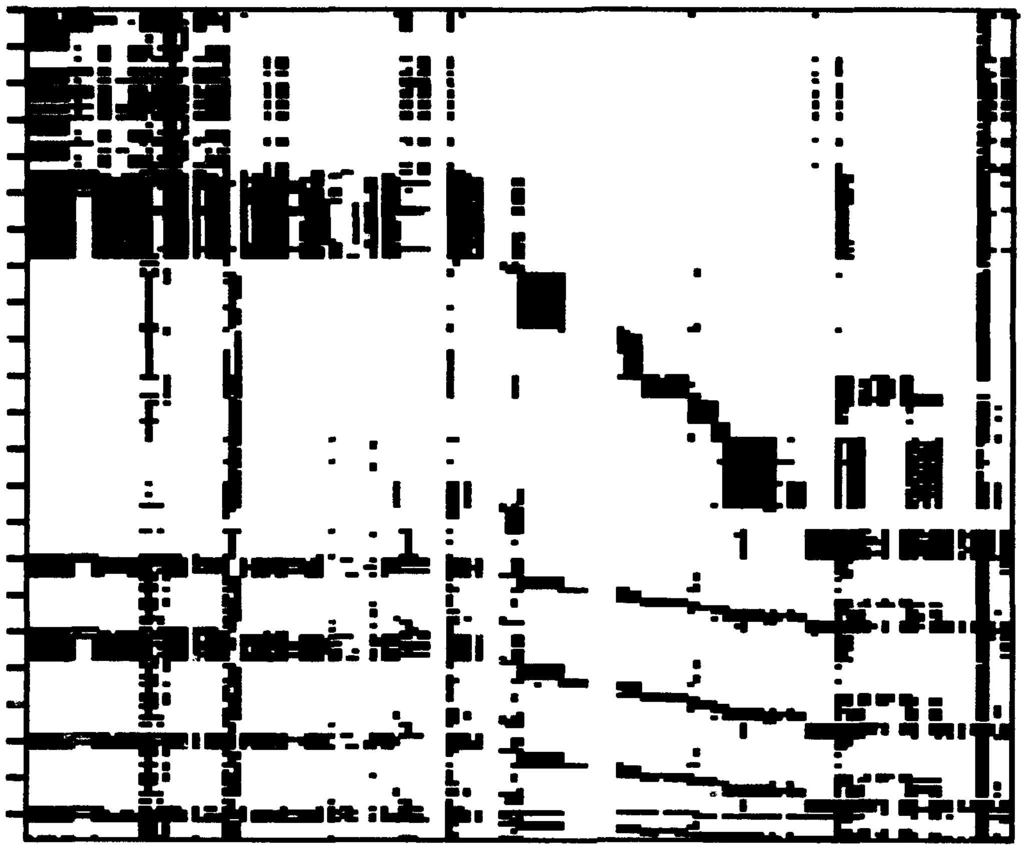

Figure 7-14: *gcc* Instruction Memory Access Patterns

This figure is a graphical representation of user instruction reference patterns for *gcc*. The *x* axis represents data addresses, ranging from 0x400000 on the left to 0x4a7000 on the right. The *y* axis represents time in terms of instructions, with instruction 0 at the top and instruction 22700000 at the bottom of the figure. Each dark point in the figure represents one or more references to a memory page during a 100000 instruction interval. This figure shows that instruction references for *gcc* are spread sparsely across the text segment, making it difficult to apply large TLB pages without substantial memory fragmentation.

SPECmarks. The large number of pages referenced during a single 100000 instruction interval illustrates clearly why *gcc* places high demands on the TLB. If variable size memory pages were to be used to improve *gcc* performance, the only solution would be to load the entire program text into a contiguous segment.

For instruction references, compilers might use feedback information for performance critical applications to locate active text contiguously, making the use of a single larger TLB entry a more attractive option. Such techniques are more difficult to apply to data references, as heap allocated structures are allocated dynamically, and so their location is not under the control of the compiler. With the relocatable nature of lisp data, it might be possible to tune garbage collectors to improve the locality of reference, although research indicates significant limitations to the benefit of such techniques [69]. For uncollected memory allocation schemes, a tool using feedback information could make suggestions of how to order heap data allocation to improve contiguity of data.

7.5. Conclusions

This chapter has investigated the performance of one and two-level instruction TLBs, data TLBs, and shared TLBs, the impact of system activity on these structures, as well as potential performance implications of variable-sized pages. This work has concentrated on fully-associative TLB organizations and split instruction and data reference streams.

For instruction TLBs, programs such as *gcc* and *lisp* that make many nested calls to small procedures are the hardest to satisfy. For most of the SPECmarks, 4K byte pages and a two entry micro-TLB (whose misses are serviced in several cycles by a shared TLB) perform reasonably well. For example, with a 3 cycle micro-TLB miss penalty (i.e., assuming that the reference hits in the 2nd-level TLB) all SPECmarks except *gcc* and *lisp* incur a CPI of less than 0.03 due to microTLB misses. *gcc* and *lisp* can achieve this level of performance with 4-entry micro-TLBs, but incur a CPI penalty of about 0.06 with a 2-entry micro-TLB. A FIFO replacement policy performs almost as well as LRU for micro-TLBs.

In single-level instruction, data, and shared TLBs, TLB performance is usually dominated by how much memory is mapped. Single-level fully-associative instruction TLBs (or the second level of a two-level organization) with more that 32 entries, 4K byte pages, and a 100 cycle miss penalties incur CPIs of under 0.1 even for *gcc*. Performance for other benchmarks and with larger TLBs is better. With the larger capacities and miss penalties of full size instruction TLBs, multi-tasking and system effects also become important.

A data or shared TLB mapping 256K bytes in 4K byte pages (i.e., 64 entries) with 100 cycle miss penalty incurs 0.1 CPI or less for all of the workloads except *nasa7* and *mat300*. Both of these are scientific/vector oriented programs with large data sets. Furthermore, column access (i.e., non-unit stride) can result in successive data references to successive pages, disastrous for TLB performance unless the entire data set is mapped at the same time. *nasa7* and *mat300* incur a CPI of 1.7 and 4.9, respectively, for the TLB parameters given above. This is not reduced to under 0.1 CPI for *mat300* until the TLB can map 2 megabytes (e.g., 256 entry TLB with 8K byte pages). Work with more demanding workloads suggests that future TLBs must map significantly more memory. Compilation techniques such as *blocking* can sometimes be applied to address this problem. Unfortunately, blocking applies only to a restricted class of scientific computations, and similar techniques for pointer-based heap structures such as occur in C programs are probably not possible.

In a model where most operating system text and data is referenced with little TLB overhead, system effects generally cause a reduction in the TLB miss rate, by amortizing user-level miss activity over a larger number of instructions.

One way to increase the amount of memory mapped without requiring an unreasonably large number of TLB entries is the use of variable-sized pages. Memory access plots suggest that the use of very large pages (e.g., 256K byte or greater) for the data space of *mat300* and the instruction space of *gcc* could vastly reduce the size of the TLB required for good performance while decreasing its miss rate.

Chapter 8
Conclusions

This thesis has explored issues in operating system structure and how structure affects system behavior and overall performance at the level of the hardware-software interface. Identifying and isolating these structural effects has required a comprehensive examination of software behavior as related to performance. This examination and the comparison of two operating systems revealed that system policy is also important and has significant impact on overall performance.

This chapter is organized as three major sections. First is a high-level summary of the major research results. This is followed by an evaluation which details some strengths and limitations of this research work and of the systems that were studied. The chapter closes with some implications for current and future work.

8.1. Summary

A major component of this research effort was the development of a set of tools for collecting and analyzing traces of memory references, to permit complete and detailed measurements of software behavior. This work has demonstrated that software instrumentation methods can be applied to collecting address traces of system behavior. The experimental results from this thesis are evidence of the utility of the approach. End-to-end measurements [71] using simulations to predict program execution time and TLB miss counts demonstrate that the trace data and simulated behavior provides an accurate models of behavior in uninstrumented systems.

Mach 3.0 and Ultrix structure differs in important ways. The comparative study of the two systems has served to identify and organize the structural differences that have the greatest impact on performance:

- **Microkernel vs. Monolithic system structure:** The microkernel structure of Mach 3.0 has significant performance impact both in terms of memory behavior and raw system instruction counts. Prior studies identified a number of specific activities which induce increased overhead in Mach 3.0 and are related to its microkernel structure. These include communication, system call emulation, context switches, TLB behavior, and increased hardware trap activity. This work has confirmed prior results, but it goes further by placing these activities in the context of overall behavior and in demonstrating that no single activity dominates the performance difference between Mach 3.0 and Ultrix. Performance penalties are distributed over a diverse range of inter-related activities, and optimization of any single activity in isolation will not have a significant impact on overall performance.
- **UNIX Implementation:** In Mach 3.0, the UNIX API is implemented in terms of Mach primitives rather than being implemented directly. This leads to increased communication and higher instruction counts.
- **Functional Differences:** Mach 3.0 provides significant additional functionality beyond that of the UNIX interface, an example being the Mach 3.0 virtual memory system. These differences lead higher instruction counts in Mach.
- **Machine Independent Code:** Mach 3.0 is a portable operating system. Ultrix is largely machine dependent, with machine dependent and machine independent code frequently mixed within procedures, and isolated by directives to the C macro preprocessor. Machine dependent code in Mach is isolated in separate files and accessed through procedure call interfaces. Some Mach functionality uses a machine-independent implementation when a machine-dependent implementation might be more efficient. These differences lead to higher instruction counts in Mach.

The structural differences between Mach 3.0 and Ultrix have a significant impact on overall performance. Memory locality is worse for Mach 3.0, and this poor locality is exaggerated by significantly higher system instruction counts. One phenomena which is reflected throughout Mach 3.0 structure is the increased use of modular code. To understand the impact of modularity, it helps to see that contiguous code has a compact footprint in the cache, while the density of modular code can be much worse. Modular code causes more procedure calls, deeper procedure call nesting, and, as a consequence, less spatial locality. Programmers are taught to consider procedure calls free, but the performance impact of this procedure call indirection can accumulate and become important, particularly when a system with long inline procedures is compared to another that is very modular and decomposed.

System policy also has a significant impact on overall performance. These policy issues are orthogonal to system structure. In comparing Mach 3.0 and Ultrix, two aspects of system policy have significant impact: disk policy and virtual to physical page mapping policy. Given their impact on performance it is no coincidence that significant prior work can be applied to address these issues. Log structured file systems can reduce the cost of conservative disk policies such as that of Ultrix [70]. Non-volatile memory could be used to make aggressive disk policies as in Mach 3.0 more safe [8]. The problems associated with page mapping policy can be addressed with cache associativity and improved page mapping policy [47]. An important contribution of this research has been to identify these issues and place them in the context of overall performance.

Memory system behavior for X11 workloads is frequently worse than that of more traditional benchmarking workloads. Frequent client-server interactions can cause increased competition misses between client, server, and system contexts. Sparse text and large data induce elevated TLB miss rates. Instruction cache behavior of X11 workloads has important similarities to that of operating system activity. Overall, these workloads differ in important ways from traditional benchmarks. If hardware designers continue to focus on traditional benchmarks, the performance of software systems such as X11 will suffer.

8.1.1. Making generalizations from this work

This research has made a detailed comparison of Mach 3.0 and Ultrix. A natural but incorrect inference one could make is that the conclusions drawn in the context of Mach 3.0 and Ultrix can be applied immediately to all microkernel and monolithic systems. The meaning of terms like "microkernel" and "monolithic" are not precise enough to allow such inferences to be made. Generalizing the results for Ultrix and Mach 3.0 require consideration of both the systems involved and the issue in question.

This work demonstrates that the structural decomposition of operating system implementation creates the potential for a new class of performance problems that do not exist for monolithic systems. This does not mean that efficient microkernel implementations are not possible. It does mean that system builders need to learn more about partitioning systems and how it affects performance. Towards this end, several recent research projects propose new ways of partitioning systems implementations to achieve some of the engineering advantages of microkernel structure without sacrificing performance [21, 50, 51, 83].

It also means that the implementors of microkernel systems must give particular attention to performance issues. System implementors have typically used elapsed time or throughput to measure and compare system performance. This thesis has demonstrated that address traces and trace-driven simulation can do much better. They go beyond telling you how long a given operation took by giving complete and detailed information about everything that happened during the operation. Simple tools such as timing devices will always be important, but for identifying and understanding the complex interactions that occur in a multiple-address-space system these simple tools are not always good enough.

Variations on the partitioning of Mach 3.0 have been proposed [21, 50, 51]. Figure 4-2 shows that if such changes are to be effective, they must reduce the dynamic instruction count for the system. Some of the behavior identified in Figure 4-2 can be attributed to specific parts of the system. For example, the higher overhead of VM activity in Mach 3.0 is due to the extended functionality and machine independent code. Higher copy overhead for Mach 3.0 is due to the need to copy file system data between address spaces. This overhead could be reduced by moving the buffer cache of file system data out of the UNIX server and into the kernel or application address space.

Other sources of instruction overhead are related in a more fundamental way to microkernel structure. Examples are overheads from traps, IPC, and TLB faults. In these cases there is potential for reducing instruction overheads through hardware support. TLB miss rates could be reduced by using large TLB pages to map user-level system contexts. Trap, copy and IPC overheads could potentially be reduced in a machine with a single global address space.

Some structural changes will have little effect because the related code is not used very often. For example, it might be possible to move UNIX process management primitives from Mach's UNIX server to the emulation library, but this would have little impact on performance, as process creation is not frequent in normal situations.

When making generalizations from this work, specific results need to be considered individually. Positive results tend to be less generalizable. An example of a positive result is "System execution has worse locality than user execution." Increasing the sizes of caches will tend to have more of an effect on bad behavior as good behavior has less potential for improvement. By this reasoning, the observed difference between system and user locality will tend to diminish as cache sizes increase.

As an example of a negative result, this research has shown that the performance impact of cache competition between user and system activity is not significant. This result tends to generalize for changes in the memory system; if caches are made smaller and system memory behavior becomes worse the impact of competition will become even less of an issue. A larger cache would tend to decreases conflict misses, both from self-interference and competition. At the same time, as the cache becomes larger the footprint of system execution will tend to expand to fill the cache, and as a result some cache miss activity will shift from self-interference to competition. At this point, cache behavior will be relatively good. The good locality of user code and relatively infrequent system activity will continue to limit the impact of competition.

8.2. Evaluation

8.2.1. Mach 3.0 and Ultrix

The two systems compared in this study were selected because they were two common workstation operating systems in academic environments at the time the study was conceived, and because they permitted the microkernel vs. monolithic system comparison which we felt to be an important and relevant distinction for present and future systems. The movement of both of the Unix implementations towards obsolescence during the course of this study demonstrates the fast pace of change in the operating systems community. It also demonstrates the importance of this kind of research, so that future systems can be created with a complete knowledge of the strengths and weaknesses of their predecessors.

This work has shown that the structure of Mach 3.0 has a negative impact on performance. Mach 3.0 was developed as a test bed for new concepts in system design, and has been very successful in this respect. It was never a goal of Mach to give better performance than other UNIX implementations, only comparable performance with new functionality.

Furthermore, the results of this thesis confirm several important points about microkernel performance:

- Memory system overheads are not the dominant barrier in achieving microkernel performance comparable to that of monolithic systems.
- Binary compatibility through system call emulation is possible and performance can be reasonable.
- The poor performance in current microkernel systems such as Mach 3.0 cannot be addressed by improving efficiency of communication primitives.

Care must be taken when applying conclusions drawn in the context of Mach 3.0 to other microkernel systems. There are significant structural differences between Mach 3.0 and other microkernels, so performance characteristics of the other systems may be different. For example, the design decision in Mach 3.0 to put the disk buffer cache in the user level UNIX server introduces extra complexity and extra copying of disk data. As a final note on Mach 3.0 performance, when a performance critical system interface is implemented using a user level server and in terms of a different set of system primitives in a different address space, optimal performance should not be expected.

8.2.2. The DECstation 5000/200 Memory System

A limitation of this study is that we based our simulations on the DECstation 5000/200 memory system. Our motive was to focus our attention on issues relating to software structure, and avoid the temptation of becoming hardware designers. The base memory system is somewhat ordinary with no glaring misfeatures, and has provided a good platform for our software comparison.

The DECstation 5000/200 memory system has very good performance for writes. This is one strength of the memory system which represents a possible limitation in our results. The write buffer has the property that it can retire consecutive writes to the same memory page with no write buffer stalls. Another important property is that the cache line size and write width are the same, so that no memory read is required for a write-miss. A memory system without these properties could have substantially higher penalties for writes.

A possible weakness of the experimental base system as compared to other contemporary memory systems is the lack of cache associativity. Instruction cache associativity could be more beneficial for system than for user code, particularly when user code is smaller than the size of the cache, as is often the case with benchmarking workloads.

A third limitation of the experimental memory system is cache size. With two 64K byte caches, the total cache size is smaller than what is available on some current machines. An important issue to consider is how the results for 64K byte caches generalize to a larger cache or deeper cache hierarchy. Lower cache miss rates can be expected for the larger caches. If cache miss penalties were constant, then the problems addressed in this thesis would tend to become irrelevant when cache size is increased. However, cache size cannot be considered independently of cache miss penalties. With faster processor speeds the penalty for a single cache miss in terms of CPU stall cycles becomes higher. This means that improved miss rates are required to maintain current memory system performance levels, and that the issues discussed in this thesis will remain important in future systems.

8.3. Implications for Current and Future Work

8.3.1. Software

Mach 3.0 demonstrates some performance pitfalls that are possible in microkernel systems. It is important to understand that performance problems are not inherent in all microkernel systems and can be separated from the fundamental elements of microkernel design. Several current research projects are meant to address these problems. Researchers at UC Berkeley have developed a technique called *sandboxing* to isolate untrusted code when it is executed in a shared address space [83]. In another project at the University of Washington and Carnegie Mellon, system functionality is partitioned into performance critical code (to be run in the user's address space) and security-sensitive code (executed in a secure system address space) [51]. Work at the University of Utah seeks to dynamically adapt functionality and migrate activities between address spaces to improve performance [21]. Windows NT from Microsoft has support for layering and configuration of functionality which is meant to addresses these issues [29]. These research and commercial efforts show promise in resolving some of the problems of microkernel performance.

Tools exist that can reorder executable program text to improve instruction cache performance [32, 55]. Typically they use profiling information to rearrange active text so that it is contiguous and more compact. These tools could be applied for making the cache footprint of system code more compact. However, this could be counter-productive without a means to prevent collisions between compacted blocks, both within and between address spaces. Furthermore, there are a number of reasons why such tools are almost never used.

- They require profile information.
- They are awkward to use.
- They do not apply readily to data.
- They are difficult to apply with dynamically loaded libraries.
- The proliferation of memory architectures for current mass-market machines makes it impractical for a software vendor to use such tools.

These problems suggest limitations for text reordering tools, and that the related problem of cache conflict misses favors a hardware solution.

8.3.2. Hardware

Microkernel performance is only a part of the system performance problem. When compared to user execution, even monolithic systems behave poorly, and as processor speed and memory speed continue to diverge, the conditions for efficient execution of system code continue to deteriorate. One traditional approach for improving cache performance is brute force: simply use larger caches and more levels of cache hierarchy. Another popular strategy is to increase cache associativity. These strategies are limited by practical considerations such as processor cycle time, the cost of building cache memory and logic, and the limited incremental improvements that larger caches can provide.

System execution exemplifies the type of extreme software behavior that tests the limitations of current memory system designs. Compulsory and collision misses make the memory system the major bottleneck in overall performance. Compulsory misses can be partially addressed with larger line sizes, but prefetch hardware [45] might eventually be required. Collision misses can be reduced with associativity and with larger caches, but as cache sizes grow, these techniques become more and more expensive.

User activity, when measured in isolation, gives a biased view of memory system behavior. As compared to user execution, the locality of system code and data is inherently poor, and hardware changes that help application execution by taking advantage of locality will not bring proportional improvements overall. Memory reference behavior in large applications has many similarities to system execution. With growing demand for good performance for these applications, hardware designers will be motivated to improve support for memory reference patterns with poor locality.

8.4. Conclusion

This thesis has explored how the structure of software systems affects the overall performance of uniprocessor computers. It has shown how interaction between the operating system and the memory system causes poor memory system performance. With the measurement tools and the experimental results presented in this research, computer systems designers will be better equipped to address the performance issues of future computer systems.

References

1. Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young. Mach: A New Kernel Foundation for Unix Development. Proceedings of the Summer 1986 USENIX Conference, July, 1986, pp. 93-113.

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

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

4. Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Kluwer Academic Publishers, Boston, MA, 1989.

5. Digital Equipment Corporation. Digital's 21064 Microprocessor. Data sheet.

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

7. Ozalp Babaoglu and William Joy. Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits. The Proceedings of the 8th ACM International Symposium on Operating System Principles, December, 1981, pp. 76-86.

8. Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhaut, Margo Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. The Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October, 1992, pp. 10-22.

9. Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. Principles of Programming Languages, January, 1992.

10. Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. Mach Kernel Interface Manual. Draft, Department of Computer Science, Carnegie Mellon Univeristy.

11. Joel F. Bartlett. SCHEME->C: A Portable Scheme-to-C Compiler. WRL Research Report 89/1, Digital Equipment Corporation Western Research Laboratory, 1989.

12. Brian N. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems. The Proceedings of the First USENIX Microkernels and Other Kernels Workshop, April, 1992, pp. 204-211.

13. Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska and Henry M. Levy. "Lightweight Remote Procedure Call". *ACM Transactions on Computer Systems 8*, 1 (February 1990), pp. 37-55.

14. Brian N. Bershad, J. Bradley Chen, Dennis Lee, and Theodore H. Romer. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. Submitted for publication.

15. Brian N. Bershad, Richard P. Draves, and Alessandro Forin. Using Microbenchmarks to Evaluate System Performance. The Proceedings of the Third Workshop on Workstation Operating Systems, April, 1992, pp. 148-153.

16. 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.

17. Joseph Boykin. *Programming under Mach.* Addison-Wesley, Reading, MA, 1993.

18. J.R. Boyton, S.L. Chakrabarti, S.P. Hiebert, J.J. Lang, J.R. Owen, K.A. Marchington, P.R. Robinson, M.H. Stroyan, J.A. Waitz. "Sharing Access to Display Resources in the Starbase/X11 Merge". *Hewlett Packard Journal 40*, 6 (December 1989), 20-32.

19. Kenneth H. Bronstein, David J. Sweetser, and William R. Yoder. "System Design for Compatibility of a High-Performance Graphics Library and the X Window System.". *Hewlett Packard Journal 40*, 6 (December 1989), 6-10. .

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

21. John B. Carter, Brian Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Lawy, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A Tool for Building Efficient and Flexible Systems. The Proceedings of the Fourth Workshop on Workstation Operating Systems, October, 1993.

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

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

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

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

26. 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 (February 1985), 270-301.

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

28. Eric C. Cooper and Richard P. Draves. C Threads. Tech. Rept. CMU-CS-88-154, Carnegie-Mellon University, School of Computer Science, February, 1988.

29. Helen Custer. *Inside Windows NT.* Microsoft Press, Redmond, WA, 1993.

30. H. Davis, S.R. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. Proceedings of the International Conference on Parallel Processing, August, 1991, pp. 99-107.

31. M. DeMoney, J. Moore, and J. Mashey. Operating System Support on a RISC. Proceedings of the 31st Computer Society International Conference (Spring Compcon '86), March, 1986, pp. 138-143.

32. Digital Equipment Corporation. *cord(1)*. Ultrix manual page.

33. D.W. Dobberpuhl, R.T. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R.A. Conrad, D.E. Dever, B. Gieseke, S.M.N. Hassoun, G.W. Hoeppner, K. Kuchler, M. Ladd, B.M. Leary, L. Madden, E.J. McLellan, D.R. Meyer, J. Montanaro, D.A. Priore, V. Rajagopalan, S. Samudrala, S. Santhanam. "A 200Mhz 64 bit Dual-Issue CMOS Microprocessor". *IEEE Journal of Solid-State Circuits 27*, 11 (November 1992), 1555-67.

34. Richard P. Draves, Brian N. Bershad, Richard F. Rashid and Randall W. Dean. Using Continuations to Implement Thread Management and Communications in Operating Systems. Proceedings of the 13th ACM Symposium on Operating Systems Principles, October, 1991, pp. 122-136.

35. Richard P. Draves. A Revised IPC Interface. Proceedings of the First Mach USENIX Workshop, October, 1990, pp. 101-121.

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

37. Alessandro Forin, David B. Golub, and Brian N. Bershad. An I/O System for Mach 3.0. Proceedings of the Usenix Mach Symposium, November, 1991.

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

39. Aaron Goldberg and John Hennessy. MTOOL: A Method for Detecting Memory Bottlenecks. WRL Technical Note TN-17, Digital Equipment Corporation Western Research Laboratory, 1990.

40. Aaron J. Goldberg and John L. Hennessy. "MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications". *IEEE Transactions on Parallel Processing 4*, 1 (January 1993), 28-40.

41. David B. Golub, Randall W. Dean, Alessandro Forin and Richard F. Rashid. UNIX as an Application Program. Proceedings of the Summer 1990 USENIX Conference, June, 1990, pp. 87-95.

42. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Palo Alto, CA, 1990.

43. Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* Ph.D. Thesis, University of California at Berkeley, Computer Sciences Division, November 1987. Number UCB/CSD 87/381.

44. Norman P. Jouppi. Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU. Proceedings of the 16th Annual International Symposium on Computer Architecture, May, 1989, pp. 281-289.

45. Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. Proceedings of the 17th Annual International Symposium on Computer Architecture, May, 1990, pp. 364-373.

46. Gerry Kane. *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1987.

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

48. James R. Larus. "Abstract Execution: A Technique for Efficiently Tracing Programs". *Software Practices and Experience 20*, 12 (December 1990), 1241-1258.

49. James R. Larus. personal communication.

50. Jay Lepreau, Mike Hibler, Bryan Ford, Jeffrey Law, and Douglas Orr. In-Kernel Servers on Mach 3.0: Implementation and Performance. Proceedings of the Third USENIX Mach Symposium, April, 1993, pp. 39-56.

51. Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. Proceedings of the 14th ACM Symposium on Operating System Principles, December, 1993, pp. 244-255.

52. Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. MemSpy, Analyzing Memory System Bottlenecks in Programs. Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, June, 1992, pp. 1-12.

53. Joel McCormack. Writing Fast X Servers for Dumb Color Frame Buffers. WRL Research Report 91/1, Digital Equipment Corporation Western Research Laboratory, 1991.

54. Joel McCormack and Bob McNamara. A Smart Frame Buffer. WRL Research Report 93/1, Digital Equipment Corporation Western Research Laboratory, 1993.

55. Scott McFarling. Program Optimization for Instruction Caches. The Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1989, pp. 183-191.

56. Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. "The MIPS R4000 Processor". *IEEE Micro 12*, 2 (April 1992), 10-22.

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

58. Jeffrey C. Mogul. SPECmarks Are Leading Us Astray. The Third Workshop on Workstation Operating Systems, April, 1992, pp. 160-161.

59. David Nagle, Richard Uhlig, and Trevor Mudge. Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures. University of Michigan, November, 1992. CSE-TR-147-92.

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

61. Michael J. K. Nielsen. Titan Systems Manual. Research Report WRL Research Report 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

62. John K. Ousterhout, G. Hamachi, Robert Mayo, W. Scott, and G.S. Taylor. "The Magic VLSI Layout System". *IEEE Design and Test of Computers 2*, 1 (February 1985), 19-30.

63. John K. Ousterhout. Why Operating Systems Aren't Getting Faster As Fast As Hardware. Proceedings of the Summer 1991 USENIX Conference, June, 1991, pp. 247-256.

64. Sharon E. Perl. *Performance Assertion Checking*. Ph.D. Thesis, Massachusetts Institute of Technology, September 1992. MIT/LCS/TR-551.

65. J.L. Peterson. XSCOPE: A Debugging and Performance Tool for X11. Proceedings of the IFIP 11th World Computer Congress, September, 1989, pp. 49-54.

66. Steven A. Przybylski. *Cache Design: A Performance-Directed Approach.* Morgan-Kaufmann, San Mateo, CA, 1990.

67. 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. Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1987, pp. 31-39.

68. Matthew H. Reilly. *Implementation and Evaluation of a Hardware/Software Performance Monitor for Parallel Programs*. Ph.D. Thesis, Carnegie Mellon University, 1989. Department of Electrical and Computer Engineering.

69. Mark B. Reinhold. *Cache Performance of Garbage-Collected Languages.* Ph.D. Thesis, Massachusetts Institute of Technology, June 1993.

70. Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. The Proceedings of the Thirteenth ACM Symposium on Operating System Principles, October, 1991, pp. 1-15.

71. J.H. Saltzer, D.P. Reed, and D.D. Clark. "End-to-End Arguments in System Design". *ACM Transactions on Computer Systems* 2 (November 1984), 277-278.

72. John Shakershober. personal communication.

73. Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating Systems Concepts.* Addison-Wesley, Reading, MA, 1991.

74. Alan Jay Smith. "Cache Memories". *ACM Computer Surveys 14*, 3 (September 1982), 473-530.

75. Micheal D. Smith. Tracing with Pixie. Stanford University, November, 1991.

76. *SPEC Benchmark Suite Release 1.0.* System Performance Evaluation Cooperative, 1989.

77. Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hal!, Englewood Cliffs, NJ, 1992.

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

79. *ULTRIX Documentation Overview for RISC Processors.* ULTRIX Documentation Group, Digital Equipment Corporation, 1989. Order number AA-NE13A-TE.

80. *UNIX User's Manual, Reference Guide* . USENIX Association, 1984. ps(1) manual page.

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

82. Bart C. Vashaw. *Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors.* Ph.D. Thesis, Carnegie Mellon University, 1992. Department of Electrical and Computer Engineering.

83. Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. Proceedings of the Fourteenth Symposium on Operating Systems Principles, December, 1993.

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

85. David W. Wall. Systems for Late Code Modification. In *Code Generation --- Concepts, Tools, Techniques*, Springer-Verlag, 1992, pp. 275-293.

86. David A. Wood, et. al. An In-Cache Address Translation Mechanism. The 13th Annual Symposium on Computer Architecture, IEEE Computer Society Press, June, 1986, pp. 358-365.

87. David A. Wood. *The Design and Evaluation of In-Cache Address Translation.* Ph.D. Thesis, Department of Computer Science, UC Berkeley, March 1991. Report Number UCB/CSD 90/565.

Appendix A

Hardware Specifications

This appendix contains hardware specifications for the equipment used in this study.

DECstation 5000 Model 200 Workstation
KN02 System Module
- MIPS R3000 central processing unit
  - 25 MHz, version 2.0, implementation 2
- MIPS R3010 floating point unit
  - version 2.0, implementation 3
- MIPS R3220 six-stage write/memory buffer
- 64 kilobyte physical instruction cache
- 64 kilobyte physical write-through data cache
- Turbochannel options:
  - 8 bit video (slot 0)
  - high resolution timer (slot 1)
  - DEC Lance thick wire Ethernet (slot 2)
- 192 megabytes of main memory in 32 megabyte SIMs

Mouse	DEC VSXXX-AA
Keyboard	DEC LK201

SCSI devices:
- Disk drives: BoxHill TD1-BDH, HP 97549T100 9136
  - Two disks were used, one for Mach 3.0, one for Ultrix
  - Sizes and bases are in 512 byte sectors.

rz0: Ultrix
partition table:

```
partition        base       size     mounted on
    a               0     112640     /
    b          112640     522240     (swap)
    e          634880    1319936     /usr
```

rz1: Mach 3.0
partition table:

```
partition         base        size    mounted on
    a                0      102400    /
    d           102400       81920    /sys
    e           184320     1770496    /usr
```

Tape Drive: Transitional Technology Inc. CTS-8210

Appendix B

Software Specifications

This appendix contains software specifications for systems used in this study.

COMPILERS:

C: DEC Ultrix C compiler version 2.1

Fortran: DEC Ultrix Fortran version 2.1

TEXT SIZES:

	text	data	bss	dec	hex
Mach 3.0					
mach_kernel.MK78.KTRACE					
normal	540880	69904	64176	674960	a4c90
instrumented	1210112	70992	64304	1345408	148780
vmunix.UX39.STDAFS+WS.epo					
normal	708608	53248	80400	842256	cda10
instrumented	1605632	53248	76208	1735088	1a79b0
emulator.UX39					
normal	57344	8192	0	65536	10000
instrumented	118784	8192	0	126976	1f000
Ultrix:					
vmunix					
normal	1323424	162928	1307632	2793984	2aa200
instrumented	2818096	162928	1847136	4828160	49ac00
Workloads:					
sed					
normal	36864	8192	33712	78768	133b0
instrumented	77824	8192	33712	119728	1d3b0
egrep					
normal	24576	8192	42560	75328	12640
instrumented	53248	8192	42560	104000	19640
yacc					
normal	49152	12288	153840	215280	348f0
instrumented	102400	12288	153840	268528	418f0
gcc					
normal	688128	102400	21472	812000	c63e0
instrumented	1515520	102400	21472	1639392	1903e0
compress					
normal	24576	8192	414496	447264	6d320
instrumented	57344	8192	414496	480032	75320
espresso					
normal	188416	20480	5600	214496	345e0

```
 instrumented   413696  20480   5600    439776  6b5e0
lisp
 normal         73728   16384   0       90112   16000
 instrumented   172032  16384   0       188416  2e000
eqntott
 normal         40960   12288   270320  323568  4eff0
 instrumented   94208   12288   270320  376816  5bff0
fpppp
 normal         167936  16384   139408  323728  4f090
 instrumented   323584  16384   139408  479376  75090
doduc
 normal         192512  20480   78704   291696  47370
 instrumented   372736  20480   78704   471920  73370
liv
 normal         24576   8192    47040   79808   137c0
 instrumented   49152   8192    47040   104384  197c0
tomcatv
 normal         61440   12288   3704144 3777872 39a550
 instrumented   139264  12288   3704144 3855696 3ad550

Andrew Benchmark components:
ar
 normal         53248   12288   7728    73264   11e30
 instrumented   110592  12288   7728    130608  1fe30
cp
 normal         20480   8192    5984    34656   8760
 instrumented   40960   8192    5984    55136   d760
du
 normal         16384   8192    7664    32240   7df0
 instrumented   36864   8192    7664    52720   cdf0
find
 normal         69632   12288   8416    90336   160e0
 instrumented   163840  12288   8416    184544  2d0e0
gcc (cc1)
 normal         1011712 155648  48496   1215856 128d70
 instrumented   2203648 155648  48496   2407792 24bd70
gcc (cpp)
 normal         114688  20480   17024   152192  25280
 instrumented   237568  20480   17024   275072  43280
gcc (gcc)
 normal         57344   16384   2240    75968   128c0
 instrumented   122880  16384   2240    141504  228c0
gcc (ld)
 normal         28672   8192    2560    39424   9a00
 instrumented   65536   8192    2560    76288   12a00
gcc (mips-tdump)
 normal         28672   12288   3120    44080   ac30
 instrumented   61440   12288   3120    76848   12c30
gcc (mips-tfile)
 normal         49152   16384   12784   78320   131f0
 instrumented   102400  16384   12784   131568  201f0
gnu-make
 normal         204800  36864   14944   256608  3ea60
 instrumented   471040  36864   14944   522848  7fa60
ls
 normal         69632   12288   56352   138272  21c20
 instrumented   159744  12288   56352   228384  37c20
```

```
mkdir
 normal          12288   8192    0       20480   5000
 instrumented    28672   8192    0       36864   9000
rm
 normal          16384   8192    0       24576   6000
 instrumented    36864   8192    0       45056   b000
sh
 normal          40960   8192    112     49264   c070
 instrumented    98304   8192    112     106608  1a070
wc
 normal          16384   8192    0       24576   6000
 instrumented    36864   8192    0       45056   b000

X11 Workload components:
X-server
 normal          1798144 110592  52528   1961264 1ded30
 instrumented    2072576 49152   26448   2148176 20c750
x11perf
 normal          274432  69632   15856   359920  57df0
 instrumented    606208  69632   15856   691696  a8df0
splot
 normal          278528  61440   339040  679008  a5c60
 instrumented    593920  61440   339040  994400  f2c60
ghostscript
 normal          946176  131072  21632   1098880 10c480
 instrumented    2113536 131072  21632   2266240 229480
```

SYSTEMS:

Both systems were compiled with the standard load order and the standard optimizations and compiler flags. Some device drivers in Ultrix are compiled without optmization.

Ultrix: Version 4.2 Revision 96

compiler flags:

```
-EL -c -G 8 -O2 -g3 -DKTRACE -DDS5000 -DDS3100 -DUWS -DDLI
-DSYS_TRACE -DRPC -DUFS -DNFS -DINET -DQUOTA -DMIPS -DKERNEL
-DTIMEZONE=300 -DDST=1 -DMAXUSERS=64 -DMAXUPRC=64
-DPHYSMEM=480 -DNCPU=1 -DBUFCACHE=10
-DKGDB -DATR_KTRACE -DATR_FLAG
```

command line for /bin/ld:

```
/bin/ld -r -EL -N -G 8 -T 80030000 -e start -o vmunix.rr
entry.o param.o ioconf.o scb_vec.o gfs_data.o dli_bind.o dli_close.o
dli_data.o dli_fetchbind.o dli_getopt.o dli_if.o dli_init.o
dli_input.o dli_open.o dli_output.o dli_proto.o dli_setopt.o
dli_timer.o dli_usrreq.o dli_subr.o gfs_bio.o gfs_descrip.o
gfs_dsort.o gfs_err.o gfs_fio.o gfs_gnode.o gfs_gnodeops.o
gfs_kernquota.o gfs_mount.o gfs_namecache.o gfs_namei.o gfs_quota.o
gfs_quotasubr.o gfs_syscalls.o gfs_sysquota.o gfs_xxx.o af.o
conf_net.o if.o if_loop.o if_to_proto.o pfilt.o gw_screen.o
gw_screen_data.o raw_cb.o raw_usrreq.o route.o net_common.o
if_ether.o in.o in_pcb.o in_proto.o ip_icmp.o ip_if.o ip_input.o
ip_output.o ip_screen.o raw_ip.o tcp_debug.o tcp_input.o tcp_output.o
tcp_subr.o tcp_timer.o tcp_usrreq.o udp_usrreq.o nfs_gfsops.o
nfs_server.o nfs_subr.o nfs_vfsops.o nfs_vnodeops.o nfs_xdr.o
vfs_dnlc.o vnodeops_gfs.o auth_kern.o auth_none.o authunix_prot.o
clnt_kudp.o klm_lockmgr.o klm_kprot.o kudp_fastsend.o pmap_kgetport.o
pmap_prot.o rpc_prot.o subr_kudp.o svc.o svc_auth.o svc_auth_unix.o
svc_kudp.o xdr.o xdr_array.o xdr_mbuf.o xdr_mem.o xdr_reference.o
mountxdr.o fifo_gnodeops.o spec_subr.o spec_vnodeops.o
kern_lock_data.o crashdump.o crash_data.o init_main.o init_sysent.o
audit_data.o kern_acct.o kern_clock.o kern_cpu.o kern_errlog.o
kern_exec.o kern_exit.o kern_fork.o kern_lmf.o kern_lock.o
kern_mman.o kern_psubr.o kern_proc.o kern_prot.o kern_resource.o
kern_sig.o kern_subr.o kern_synch.o kern_time.o kern_utctime.o
kern_tpath_data.o kern_xxx.o subr_prf.o subr_rmap.o subr_xxx.o
syscalls.o sys_generic.o sys_process.o sys_socket.o sys_sysinfo.o
sys_trace.o tty.o tty_conf.o tty_pty.o tty_subr.o tty_tb.o tty_pcm.o
tty_tty.o uipc_domain.o uipc_mbuf.o uipc_msg.o uipc_pipe.o
uipc_proto.o uipc_sem.o uipc_smem.o uipc_socket.o uipc_socket2.o
uipc_sysV.o uipc_syscalls.o uipc_usrreq.o vm_drum.o vm_kmalloc.o
vm_mem.o vm_mon.o vm_page.o vm_proc.o vm_swalloc.o vm_sched.o
vm_smem.o vm_subr.o vm_sw.o vm_swap.o vm_swp.o vm_text.o ws_device.o
ufs_alloc.o ufs_bmap.o ufs_flock.o ufs_gnode.o ufs_gnodeops.o
ufs_mount.o ufs_namei.o ufs_subr.o ufs_syscalls.o ufs_tables.o
ufs_xxx.o pseudo_data.o kgdb.o kgdb_stub.o sysmips.o cpuconf.o
cons_char.o autoconf.o cache.o check_dbg.o conf.o cons_sw.o
coproc_control.o debug.o emulate_instr.o fp_intr.o hwconf.o kn01.o
```

```
kn02.o kn230_copy.o mc146818clock.o clock.o mdc.o in_cksum.o kopt.o
locore.o machdep.o mem.o panic.o process.o softfp.o softfp_unusable.o
startup.o swtch.o sys_machdep.o tlb.o trap.o ufs_machdep.o usercopy.o
vec_intr.o vm_machdep.o sm_machdep.o pt_machdep.o if_ln.o
if_ln_copy.o if_ne.o if_uba.o if_fza.o scsi.o scsi_disk.o scsi_tape.o
scsi_sii.o scsi_asc.o pdma_ds5000.o pdma3min.o pdma_entry.o
pdma_func.o vba.o vbainit.o xviainit.o xvmeinit.o vbavar.o
vba_errors.o vme_routines.oautoconf_data.o af_data.o cons_sw_data.o
gx_data.o ga_data.o dc_data.o if_ln_data.o if_ne_data.o if_fza_data.o
if_to_proto_data.o tc_option_data.o scsi_data.o mdc_data.o
tty_conf_data.o tty_pty_data.o uipc_domain_data.o vba_data.o
tc.odc7085.o scc.o gx.o gq.o ga.o qfont.o xcons.o fb.o bt459.o
bt455.o bt431.o pmagaa.o pmvdac.o lk201.o vfb03.o fb_data.o
lk201_data.o mmap_data.o ktrace.o atrace.o ktr_cprocs.o ktr_sprocs.o
vers.o swapgeneric.o
```

Mach 3.0 kernel:
version: MK78
config: STD+ANY+chen_atrace+fixpri

compiler flags:
```
-c -O2 -EL -G 32 -MD -DMACH -DMACH_KERNEL -DCHEN_ATRACE
-DCHEN_KTRACE -DKERNEL
```
command line for /bin/ld:
```
/bin/ld -r -o mach_kernel.MK78.KTRACE.rr -EL -G 32 -N -T 80030000
-e start start.o db_access.o db_aout.o db_break.o db_command.o
db_cond.o db_examine.o db_expr.o db_ext_symtab.o db_input.o
db_lex.o db_macro.o db_output.o db_print.o db_run.o db_sym.o
db_task_thread.o db_trap.o db_variables.o db_watch.o
db_write_cmd.o ipc_entry.o ipc_hash.o ipc_init.o ipc_kmsg.o
ipc_marequest.o ipc_mqueue.o ipc_notify.o ipc_object.o ipc_port.o
ipc_pset.o ipc_right.o ipc_space.o ipc_splay.o ipc_table.o
ipc_thread.o mach_debug.o mach_msg.o mach_port.o ast.o
bootstrap.o counters.o debug.o eventcount.o exception.o host.o
ipc_host.o ipc_kobject.o ipc_mig.o ipc_sched.o ipc_tt.o kalloc.o
lock.o mach_clock.o mach_factor.o machine.o printf.o priority.o
processor.o queue.o sched_prim.o startup.o syscall_emulation.o
syscall_subr.o syscall_sw.o task.o thread.o thread_swap.o
time_stamp.o timer.o xpr.o zalloc.o memory_object_data_provided.o
memory_object_data_unavailable.o memory_object_data_error.o
memory_object_set_attributes.o memory_object_data_supply.o
memory_object_ready.o memory_object_change_attributes.o
mach_host_server.o mach_port_server.o mach_server.o
memory_object_default_user.o memory_object_user.o
mach_debug_server.o memory_object.o vm_debug.o vm_external.o
vm_init.o vm_kern.o vm_map.o vm_object.o vm_pageout.o
vm_resident.o vm_user.o blkio.o chario.o cirbuf.o dev_lookup.o
dev_name.o dev_pager.o device_reply_user.o device_server.o
device_init.o ds_routines.o net_io.o subrs.o ioconf.o vm_fault.o
autoconf.o conf.o context.o db_disasm.o db_interface.o
db_mips_sym.o db_trace.o locore.o mips_cache.o mips_copyin.o
mips_cpu.o mips_init.o mips_instruction.o mips_mem_ops.o
mips_misc.o mips_startup.o parse_args.o pcb.o pmap.o
prom_interface.o softfp.o swapgeneric.o tlb.o trap.o bt431.o
bt455.o bt459.o bt478.o busses.o cfb_hdw.o cfb_misc.o dc503.o
dtop_handlers.o dtop_hdw.o dz_hdw.o ecc.o fb_hdw.o fb_misc.o
fdc_82077_hdw.o frc.o ims332.o isdn_79c30_hdw.o kernel_font.o
lance.o lance_mapped.o lk201.o mc_clock.o mouse.o pm_hdw.o
pm_misc.o scc_8530_hdw.o screen.o screen_switch.o
serial_console.o xcfb_hdw.o xcfb_misc.o ga_hdw.o ga_misc.o
gq_hdw.o gq_misc.o gx_misc.o kmin.o kmin_cpu.o kmin_dma.o kn01.o
kn02.o kn02_dma.o kn02ba.o maxine.o maxine_cpu.o mips_box.o
model_dep.o tc.o mapped_scsi.o rz.o rz_cpu.o rz_disk.o
rz_disk_bbr.o rz_host.o rz_labels.o rz_tape.o scsi.o
scsi_53C94_hdw.o scsi_7061_hdw.o scsi_alldevs.o scsi_comm.o
scsi_cpu.o scsi_disk.o scsi_jukebox.o scsi_optical.o
```

```
scsi_printer.o scsi_rom.o scsi_scanner.o scsi_tape.o scsi_worm.o
atrace.o ktrace.o atrmem.o vers.o
```

Mach 3.0 UNIX server:

version: UX39

config: STDAFS+WS

compiler flags:

```
-02 -Wf,-XNk150 -EL -G 32 -MD -DCMU -DINET -DMACH
-DPMAX -DKERNEL
```

command line for /bin/ld:

```
ld -o vmunix.UX39.STDAFS+WS.normal -EL -G 32 -e __start
crt0.o afs_buffer.o afs_cache.o afs_call.o afs_callback.o
afs_daemons.o afs_dir.o afs_gateway.o afs_istuff.o afs_lock.o
afs_osi.o afs_osifile.o afs_osinet.o afs_physio.o afs_pioctl.o
afs_resource.o afs_vfsops.o afs_vnodeops.o fcrypt.o rxkad_client.o
rxkad_common.o nfs_gateway.o Kcallback.ss.o Kvice.cs.o Kvice.xdr.o
afsaux.o afsvlint.cs.o afsvlint.xdr.o cmu_syscalls.o init_sysent.o
kern_acct.o kern_descrip.o kern_mman.o kern_proc.o kern_prot.o
kern_resource.o kern_time.o kern_xxx.o mach_init.o mach_clock.o
mach_core.o mach_exec.o mach_exit.o mach_fork.o mach_process.o
mach_signal.o mach_synch.o quota_sys.o subr_log.o subr_prf.o
subr_rmap.o subr_xxx.o syscalls.o sys_generic.o sys_socket.o tty.o
tty_cmupty.o tty_conf.o tty_pty.o tty_subr.o tty_tty.o uipc_domain.o
uipc_mbuf.o uipc_proto.o uipc_socket.o uipc_socket2.o uipc_syscalls.o
uipc_usrreq.o param.o af.o if.o if_loop.o netisr.o raw_cb.o
raw_usrreq.o route.o if_ether.o igmp.o in.o in_pcb.o in_proto.o
ip_icmp.o ip_input.o ip_mroute.o ip_output.o raw_ip.o tcp_debug.o
tcp_input.o tcp_output.o tcp_subr.o tcp_timer.o tcp_usrreq.o
udp_usrreq.o nfs_server.o nfs_subr.o nfs_vfsops.o nfs_vnodeops.o
nfs_xdr.o rfs_control.o rfs_descrip.o rfs_init.o rfs_kern.o
rfs_socket.o rfs_subr.o rfs_syscalls.o rfs_ticket.o auth_kern.o
authunix_prot.o clnt_kudp.o clnt_perror.o kudp_fastsend.o
pmap_kgetport.o pmap_prot.o rpc_callmsg.o rpc_prot.o subr_kudp.o
svc.o svc_auth.o svc_auth_unix.o svc_kudp.o xdr.o xdr_array.o
xdr_mbuf.o xdr_mem.o rx.o rx_bcrypt.o rx_clock.o rx_event.o
rx_globals.o rx_kernel.o rx_null.o rx_vab.o xdr_rx.o bdev_vnodeops.o
fifo_vnodeops.o spec_subr.o spec_vnodeops.o ufs_alloc.o ufs_dsort.o
ufs_inode.o ufs_subr.o ufs_tables.o block_io.o cons.o device_misc.o
device_reply_hdlr.o device_utils.o disk_io.o ether_io.o inittodr.o
misc.o port_hash.o proc_to_task.o queue.o stubs.o syscall.o
syscall_subr.o tty_io.o user_copy.o user_reply_msg.o ux_exception.o
ux_server_loop.o zalloc.o bsd_server.o bsd_server_side.o xpr.o
ufs_bmap.o ufs_dir.o ufs_vfsops.o ufs_vnodeops.o vfs.o vfs_bio.o
vfs_conf.o vfs_dnlc.o vfs_io.o vfs_lookup.o vfs_pathname.o
vfs_syscalls.o vfs_sysnames.o vfs_vnode.o inode_pager.o bsd_machdep.o
conf.o in_cksum.o mapped_ether.o mips_exception.o misc_asm.o
mips_ptrace.o vers.o -lthreads -lmach_sa
```

Mach 3.0 emulator: version UX39

compiler flags:

```
-MD -Wf,-XNk150 -02 -g3 -DMAP_UAREA -DMAP_FILE
```
command line for /bin/ld:
```
ld -z -o emulator.UX39.out.rr -T fc00000 -D fd00000 -e __start
-r -nocount crt0.o -count bsd_user_side.o emul_init.o
emul_stack_alloc.o emul_generic.o  allocator.o syscall_table.o
emul_machdep.o emul_vector.o emul_cache.o emul_mapped.o
bsd_1_user.o -nocount -lthreads -lmach_sa btrace_emul.o
```

Appendix C

List of Abbreviations

API:	application programmer interface
CPI:	cycles per instruction
CPU:	central processing unit
I/O:	input / output
IPC:	interprocess communication
KTLB:	kernel TLB references/misses
LRU:	least recently used
MCPI:	memory cycles per instruction
RISC:	reduced instruction set computer
RPC:	remote procedure call
TLB:	translation lookaside buffer
UTLB:	user TLB references/misses
VM:	virtual memory

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890