OS/Architecture Interactions and their Influence on Computer Architecture

by

David Nagle

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1995

Doctoral Committee:

Professor Trevor Mudge, Chair
Professor Brice Carnahan
Professor Edward Davidson
Assistant Professor Stuart Sechrest
Joel Emer, Senior Consulting Engineer, Digital Equipment Corporation

DEDICATION

To mom and dad — for all of their love, encouragement, and support.

ACKNOWLEDGEMENTS

A friend may well be reckoned the masterpiece of Nature.

Ralph Waldo Emerson

Today I find myself at the end of school. It's an amazing time — a new job with lots of people to meet and tasks to accomplish. It is also a time of reflection, a time to recognize the friends who have helped me through the years. My friends have given me so much, helped me to learn and to grow and have shared their lives with me. I thank each for their counsel, friendship and love.

Hugh Cooper helped me understand the art of teaching. He also taught me that, with a bit of creativity and a lot of hard work, there is always another way to attack a problem. It is a lesson I carry with me every day of my life.

Brice Carnahan and James Wilkes gave me the opportunity to teach some of the best and brightest students. It was in their Engineering 103 courses that I realized how rewarding it is to help people learn and grow.

Trevor Mudge is my good friend and advisor. His advice and experience allowed me to keep perspective while his patience and friendship guided me through graduate school. I look forward to many more lunches at Casey's.

Stuart Sechrest showed me how rewarding research can be and the importance of brainstorming. I will miss our impromptu conversations.

The graduate students who I worked with on the GaAs MIPS project were an amazing group of people. Tom Huff, Mike Upton, Tim Stanley and Mike Riepe all helped to make graduate school an interesting adventure.

Rob Novak was my roommate through most of graduate school. While that in itself is a noble feat, he also taught me the importance of taking time for myself and showed me the meaning of generosity.

Frank walked with me through one of the most amazing years in my life. His friendship and advice have been a unique gift that I value very much.

Julie Scherer helped me to understand the importance of sharing my love and friendship. Julie also taught me to have the courage to grab hold of life and not be afraid to follow my dreams.

Sean showed me the rewards of living life with a smile and good cheer. His strength, counsel and love are a continual inspiration to me.

David Grinnell's love and counsel gave me the strength to finish this work, to start my new career, and to share my life with him.

Rich Uhlig is my research partner and best friend. His example has been my teacher. His advise has been my guide. His support has been my strength. After years of working together, our paths have diverged for a while — each beginning a new life with new friends, each gaining new experiences and learning new things. I hope, however, that our partnership and friendship will continue to grow.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

The Importance of OS/Architecture Interactions

1.1 Introduction

Operating systems are important. Services provided by or accessed through the operating system (OS) are part of almost every application. Yet, most architectural studies have not considered OS effects. A lack of tools capable of capturing OS references, coupled with the complex behavior of the operating system, has forced most researchers either to qualitatively reason about the OS or to neglect the OS altogether.

This is in stark contrast to application/architecture research, which has developed a very good quantitative analysis methodology. Today, architects build accurate models that analyze interactions between hardware and application software, measuring the impact of various designs and helping to select architectural components that deliver the best performance within the constraints of a technology. The use of these results, however, is severely limited by their inability to account for the operating system's influence on architectural performance.

This dissertation addresses this problem by extending the quantitative approach to the study of the interactions between operating systems and architectures (OS/architecture interactions). Using a hybrid hardware/software monitoring system to gain access to all hardware and software activity, we measure the performance of real systems, analyze the interactions between the OS and architecture, and explore architectural design trade-offs that improve performance. Our goal is to provide designers with an understanding of how

Benchmark	Simulated CPI	Actual CPI	% Difference Between Simulated and Actual CPI	% of Run Time Spent Outside the Benchmark
fpppp	2.49	3.05	22.5%	0.7%
gs	1.31	1.68	28.2%	48.6%
jpeg_play	1.30	1.50	15.4%	12.8%
mpeg_play	1.37	1.82	32.9%	25.6%
small	1.01	1.27	25.7%	30.0%
video_play	1.17	2.44	108.6%	69.4%

Table 1.1 Simulation vs. actual architectural performance

Performance estimates obtained from simulation tools (e.g., cache2000 and pixstats) can be very inaccurate. The absolute error in cycles per instruction (CPI) varies from 15% (jpeg_play to over 100% (video_play). The "Actual CPI" does not include the CPI contribution due to TLB misses.

To produce the simulated CPI, each benchmark was annotated to produce an address trace of every instruction and data reference. The address trace was fed into a simulator, cache2000, which models a DECstation 3100 memory system. The results from cache2000 were combined with the static analysis results from the MIPS tool pixstats to compute the "Simulated CPI" values. Actual CPI was measured from a running workstation using our Monster monitoring system (see Chapter 2).

to analyze OS/architecture interactions and how architectures can overcome current performance problems.

1.2 Why consider the operating system?

Neglecting the OS would not be a serious problem if the OS did not adversely influence the behavior and performance of a system. However, the handful of works that have measured OS/architecture interactions consistently show a performance problem over a range of applications[1] [Smith82, Clark84, Clark85, Clark88, Agarwal89, Torrellas92]. Consider the measurements in Table 1.1 (p. 2), a comparison of application-only simulation against actual machine performance, including all OS and system service

1. The exception is scientific applications, which typically utilize few OS services and can be accurately characterized without considering the operating system.

Benchmark	CPI Ultrix	CPI Mach 3.0
fpppp	3.05	2.96
gs	1.68	1.83
jpeg_play	1.50	1.51
kenbus	2.20	2.38
mpeg_play	1.82	2.00
real_gcc	1.68	1.83
sdet	1.89	2.24
video_play	2.44	2.28

Table 1.2 Architectural performance differences between Ultrix and Mach 3.0

The data show that different operating systems can have very different architectural performance. Despite the fact that the same binaries were used under both operating systems, there are significant differences in architectural performance between Ultrix and Mach 3.0. Both measurements were taken from a DECstation 3100.

activity. In each case, the simulation tool's inability to account for OS activity[1] leads to a large gap between the predicted and actual performance.

Aggravating this problem are fundamental changes in OS and software technologies. Applications are increasing their use of OS services, operating systems are providing more functionality, and advances in OS technology are fundamentally changing the behavior of the operating system and its interaction with the processor and memory architecture. Table 1.2 (p. 3) compares the actual CPI measurements of two different operating systems: an old-style monolithic OS, Ultrix, and a newer microkernel based OS, Mach 3.0. The data show a large difference in CPI, even though both measurements are from the same machine running the same set of application and X server binaries.

Neglecting the operating system clearly results in an inaccurate characterization of performance. The data in Table 1.2 demonstrates that performance can differ between

1. "OS activity" includes memory references from the OS kernel, OS servers (for Mach 3.0), system service references (such as an X display server) and multi-tasking effects.

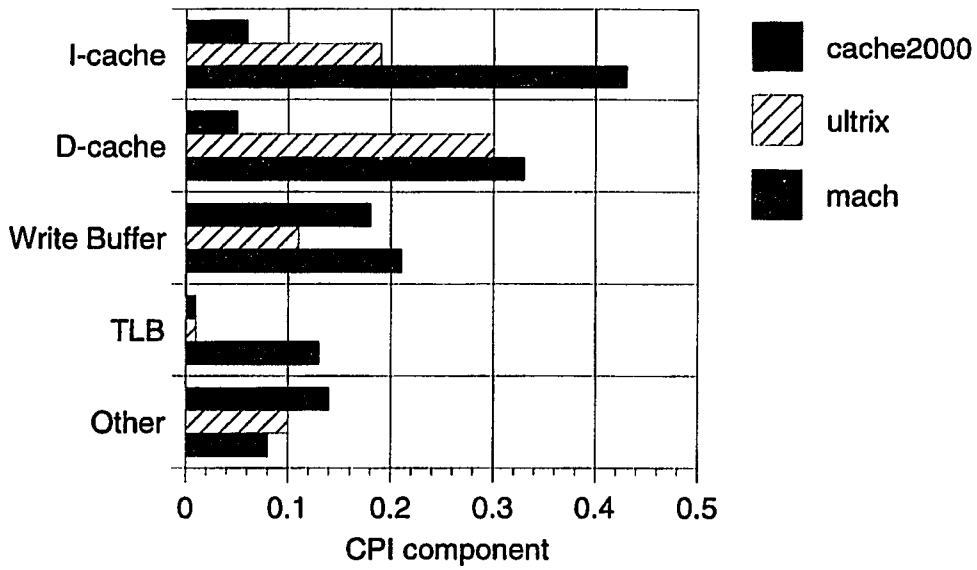


Figure 1.1 Stall breakdown by architectural component

The figure shows three different measurements of the mpeg_play workload running on a DECstation 3100.

The DECstation 3100 is a MIPS R2000 based workstation with a 64-KByte instruction cache (I-cache), 64-KByte data cache (D-cache), a 4-entry write buffer, and a 64-entry fully-associative TLB (translation lookaside buffer). Processor stalls can be due to any of these hardware structures or *other* types of stalls including integer multiply/divide, interrupts, and floating point stalls.

---

operating systems and that architects may need to consider multiple operating systems in order to fully characterize OS/architecture interactions. The results, however, do not provide any insight into where performance is lost. We examine this question in the next section.

1.3 Where is the performance going?

To answer this question, we used our system monitoring facilities (see Chapter 2) to measure the amount of time a workload spends stalling in each of the architectural components of a DECstation 3100. From these measurements, we can determine which hardware structures are the most important in terms of stall cycles. This information can

then be used to redesign the architecture to better support the software. Figure 1.1 (p. 4) shows the stall breakdown for an example[1] workload, mpeg_play.

In the cache2000 simulations, which only consider application references, the two most important architectural components are the write buffer and "other", which include integer multiply/divide stalls, floating point stalls and exception stalls. Together, write buffer and other stalls account for over 75% of all stall cycles. Using this data, an architect would find the instruction cache (I-cache) and data cache (D-cache) performance acceptable and focus on improving the write buffer and "other" performance.

Real measurements under Ultrix reveal a very different performance picture. Here, the write buffer and "other" account for less than 30% of the total stall cycles. The biggest performance problem is the D-cache, which is responsible for over 40% of the stall cycles. Combined, the I- and D-caches account for 70% of the stall cycles. In contrast, cache2000 measurements suggest that caches account for less than 26% of the stall cycles.

The Mach 3.0 measurements portray yet another picture. Like Ultrix, the I- and D-caches are responsible for the largest portion of stalls. However, the total number of stall cycles has increased and the relative importance of I- and D-caches has switched, with the I-cache now responsible for over 40% of the total stall cycles. The graph also shows a significant increase in translation lookaside buffer (TLB) stalls, from 0.01 to 0.13 CPI. This shift between D- and I-cache and the increased importance of the TLB are important differences because they can influence architectural design trade-offs.

Measurements across a range of workloads show similar results (Figure 1.2, p. 6. - see Table 2.3, p. 27, for a description of the workloads). In almost every case, I-cache stalls are the largest source of stalls under Mach 3.0. Mach 3.0 also increases the number of TLB misses over Ultrix, by 300% in the average case.

1. mpeg_play's performance characteristics are very similar to the performance characteristics averaged over our entire benchmark suite.

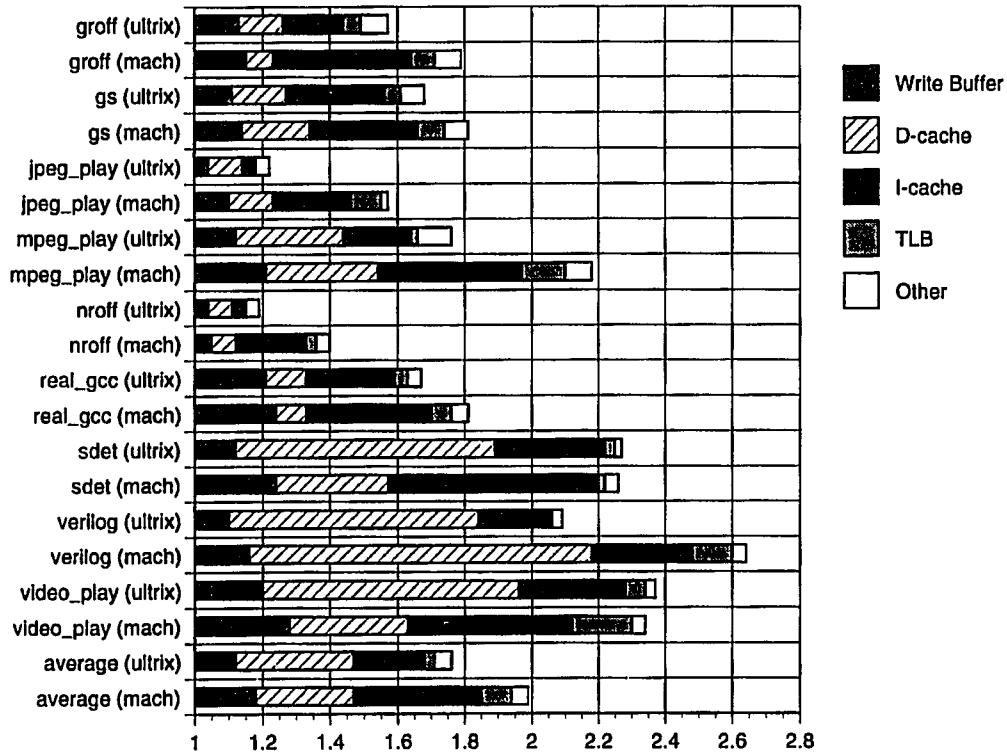


Figure 1.2 Stall breakdown for workload suite

Most of the workloads show a shift in the relative importance of D- and I-cache stalls and an increase in TLB stalls, between Ultrix and Mach.

This information is invaluable because it shows the extent to which architectural components contribute to performance loss, allowing designers to focus their attention on the important architectural issues. However, the results do not explain *why* there are significant differences between the operating systems; the results only provide a coarse-grained architectural view of the system's performance. Understanding "why" is far more important because it helps designers reason about how current and future software trends will impact the architecture. This, in turn, allows hardware designers to build architectures that meet the needs of current and future generations of software.

1.4 Analyzing OS/architecture interactions

The data in Figure 1.2 (p. 6) demonstrate that the OS can significantly impact performance. Identifying the hardware components responsible for lost performance is only the first problem. *The more difficult problem is to determine what are the source(s) of the performance loss.* In particular to understand if performance problems are due to:

- the architecture,
- the operating system,
- or some interaction(s) between the architecture and the operating system.

In the next chapter, we examine this question by analyzing the structure of the architecture, the operating system and the nature of their interactions. Our base architecture is a MIPS R2000 based DECstation 3100. The operating systems considered are three variants of UNIX with diverse internal structures: DEC Ultrix, OSF/1, and Mach 3.0.

After identifying several interactions between the operating system and architecture, we focus on OS interactions with the TLB and the I-cache. Analysis begins by identifying the reasons for TLB and I-cache performance problems. Using several different simulation techniques, we explore architectural design trade-offs to determine how much support the architecture can provide. Towards the end of our analysis, we propose a framework for studying OS/architecture interactions and use the framework to relate previous and concurrent research in the field.

The major contributions of this work are the following. First, we analyze OS/architecture interactions across several UNIX based operating systems with radically different internal organizations. Second, we explore architectural trade-offs to help alleviate the performance problems introduced by the operating system. Finally, we use

our results and the works of others to develop an OS/architecture framework that allows researchers to better analyze OS/architecture interactions.

Chapter 2 explores the performance problems and recent attempts to measure OS/architecture interactions. It also outlines our approach and applies the approach to diagnose the basic performance problems found in Ultrix, OSF/1 and Mach 3.0 running on a DECstation 3100. In Chapter 3, we begin our analysis, focusing on the translation look-aside buffer (TLB) and its interaction with the software. Chapter 4 explores caches and instruction fetching techniques. In Chapter 5, we review previous and concurrent work, describing what we have learned about OS/architecture interactions and outlining issues that remain unresolved. Conclusions and future work are presented in Chapter 6.

While we defer a full discussion of previous work to Chapter 5, we should mention that there have been other important contributions to the body of research in this area. Concurrent examples are Flanagan at Brigham Young University [Flanagan93] and Chen at Carnegie Mellon University [Chen94]. There are also several previous studies by Smith [Smith82], Clark et al. [Clark83, Clark85a, Clark85b], Agarwal [Agarwal88] and McRae [McRae93] which cover similar material. Chapter 5 will summarize both previous and concurrent work, placing issues into an OS/architecture framework that allows designers to understand how various components and design issues interact and affect performance.

CHAPTER 2
Measuring OS/Architecture Interactions

The art and science of computer performance boils down to characterizing the workload
Domenico Ferrari

2.1 Introduction

Why are OS/architecture interactions difficult to analyze? Probably the most common problem is the lack of tools capable of probing into the system. Even when the tools are available, there are a number of issues which are not usually encountered in standard application/architecture analysis. This chapter explores some of these issues. We begin by examining how recent works have attempted to analyze OS/architecture interactions, discussing some of the problems other works have encountered and the role benchmarks play in determining the results. We continue the discussion of benchmarks, examining issues that are unique to OS/architecture studies. Next, we outline a number of software and hardware tools that can be used to probe into the system. Finally, we discuss our tools and techniques, how they work and how we have applied them to the problem of analyzing OS/architecture interactions.

2.2 How should we look at the problem?

One of the major challenges in exploring the interactions between architectures and operating systems is understanding how to approach the problem. One approach is to treat the operating system as just one more piece of software. Several previous works have

adopted this *architecture-centric* viewpoint [Smith82, Clark83, Clark85a, Clark85b, Agarwal88]. Using either hardware monitoring facilities or complete address traces[1], these works show that operating system code can significantly degrade the performance of hardware structures such as TLBs, write buffers, and caches.

A more *OS-centric* approach focuses on the performance of basic OS functions and associated architectural support. Two recent works, [Ousterhout89, Anderson91], use this approach, applying software methods and qualitative analysis techniques to analyze the performance of specific architectural structures. Their results suggest that current architectural trends cause OS performance to lag behind expected performance, creating a gap between application and OS performance.

Unfortunately, neither approach provides a complete picture of how OS/architecture interactions affect overall performance. By treating the OS as just another piece of code, the architecture-centric viewpoint is unable to determine how OS policy and implementation impact the architecture. This prevents architecture-centric studies from determining if the problem is best solved in hardware, software, or some combination of the two. Likewise, by focusing on specific architectural structures, the OS-centric approach cannot determine how the architecture, as a whole, impacts OS performance.

Worse, using a one-sided approach can sometimes lead to inaccurate results and misleading conclusions. For example, Ousterhout [Ousterhout89] and Anderson et al. [Anderson91] employed an OS-centric approach to show that "the performance of OS functions has not scaled in a commensurate way" with respect to application performance. Using a set of micro-benchmarks that implemented several operating system primitives such as context switching and exception handling, both works measured the micro-benchmark performance across a range of architectures, comparing the micro-benchmarks' relative performance increases against the performance increases of the

1. These complete address traces included OS references.

Operation	DEC CVAX (µsecs)	MIPS R2000 (µsecs)	Relative Speed (R2000/CVAX)
Null system call	15.8	9.0	1.8
Trap	23.1	15.4	1.5
Page table entry change	8.8	3.1	2.8
Context switch	28.3	14.8	1.9
Application Performance	—	—	4.2

Table 2.1 Relative performance of primitive OS functions

This table, reproduced from Anderson et al. [Anderson91], compares the performance of the 11.1 MHz DEC CVAX with the 16.67 MHz MIPS R2000 processor. While the R2000's performance is better than the CVAX, the relative speed of R2000 to CVAX is significantly below that found by comparing the two machines' SPEC90 performance numbers.

SPEC90 application-based benchmark suite[1]. The data for one set of experiments is reproduced in Table 2.1[2] and was used to show how OS performance is lagging behind application performance. The implication was that architectural trends are not supporting operating system code as well as application code and that technologies such as RISC make it more difficult to implement efficient operating systems. To quote Ousterhout, "Operating systems are not getting faster as fast as hardware."

Unfortunately, the analysis did not consider important architectural issues, causing the researchers to draw several inaccurate conclusions. First, the comparison is based on the SPEC90 benchmark suite, which includes floating point and integer benchmarks. One source of significant improvement for SPEC90 is the superior floating point hardware found in many newer architectures[3]. This clearly will not benefit operating system code which executes almost no floating point operations. Second, the CVAX on-chip cache is

1. Ousterhout used the millions of instructions per second (MIPS) rating as the base for comparison. While this is different from Anderson, who uses SPEC90, architectural analysis of Ousterhout's workstations reveals similar problems in his analysis.

2. For the complete list of results, refer to Anderson et al., Table 1 [Anderson91].

too small to hold the working set of any of the SPEC90 benchmarks[1]. Therefore, we expect the R2000's large primary caches to give the SPEC90 benchmarks a significant performance improvement over the CVAX. The cache difference will not, however, provide any performance improvement for the micro-benchmarks because their working sets fit comfortably inside either the CVAX or MIPS primary cache.

With this architectural information, it becomes clear that this experiment provides little insight into OS/architecture interactions. It is true that the micro-benchmark performance did not scale at the same rate as the SPEC90 benchmarks. But, comparing the SPEC90 benchmark suite against the micro-benchmarks biased the experiment in favor of SPEC90 because it utilizes the floating point and cache improvements, while the micro-benchmarks do not.

Ironically, the data collected by Anderson can be used to answer an important question: "Are newer architectures improving the performance of OS primitives?" We answered this question by computing the number of cycles the CVAX and the MIPS architectures require to execute each primitive (Table 2.2). The results show that the MIPS architecture requires fewer cycles to execute each primitive OS function. So, while OS code may not be seeing the same rate of performance improvement as some user applications, architectural features are improving the performance of OS primitives. In other words, "operating systems are getting faster faster than hardware."

The fundamental limitation of most of the previous work has been its singular viewpoint. This is not to suggest that the previous work did not make a significant contribution. On the contrary. The VAX hardware measurement papers [Clark83, Emer84, Clark85, Clark88] clearly showed that OS code can significantly degrade the performance

3. For example, the R2000's floating point add, subtract and multiply algorithms are a full cycle faster than the CVAX.

1. The CVAX has a 1-KByte unified primary cache backed by a 64-KByte second level unified cache (2 cycle penalty on a primary cache miss that hits in the second level cache.

Operation	DEC CVAX (µsecs)	MIPS R2000 (µsecs)	DEC CVAX (cycles)	MIPS R2000 (cycles)
Null system call	15.8	9.0	176	150
Trap	23.1	15.4	257	257
Page table entry change	8.8	3.1	98	52
Context switch	28.3	14.8	314	247

Table 2.2 Number of cycles required to execute primitive OS functions

Using the data from Anderson et al. [Anderson91], this table shows the number of cycles required to execute each primitive OS function. The results show that the MIPS architecture actually requires fewer cycles to execute each primitive.

of hardware structures. Smith [Smith82] and Agarwal [Agarwal89] examined how OS code is affected by various cache design trade-offs and that neglecting OS references can result in optimistic performance simulations. Finally, while some of the quantitative analysis by Ousterhout [Ousterhout89] and Anderson et al. [Anderson91] misses the mark, their qualitative analysis of operating system and architectural trends was very insightful and has helped to guide much of the work that has been done over the last 4 years.

2.3 Benchmarking for OS/architecture interactions

Benchmarks are the basic ruler used to measure performance. However, choosing an appropriate benchmark is one of the most difficult aspects of designing a good experiment. A well-chosen set of benchmarks, carefully applied, will exercise appropriate system component(s), revealing valuable performance information. A poorly chosen set can overlook important components, focusing attention on issues of negligible importance.

Using benchmarks to construct a meaningful experiment requires a researcher to answer two basic questions.

- What specific relationship is being examined?

- Do the benchmarks exercise the system in such a way as to generate data that can elucidate this relationship?

While these questions may seem obvious, experimental work has often overlooked the second question, applying a "standard" benchmark suite without considering whether the benchmarks are appropriate for the performance issues being evaluated. For example, many published experiments only use the SPEC Floating Point and Integer Benchmark Suite to evaluate memory system designs. These benchmarks, however, do not exercise the instruction side of a memory system [SPEC91, Gee93, Uhlig95]. This limits the general applicability of the experimental results because the benchmarks are not an appropriate set of workloads for many studies.[1,2]

Researchers often overlook another important question, "Do the benchmarks bias the results of the experiment?" This is particularly true of experiments that use code fragment benchmarks (also known as micro-benchmarks) instead of real applications. Code fragments can rigorously exercise specific system component(s), but do not provide any information on the relative importance of the component(s). They can also remove the effects of hardware and/or software structures, creating artificially good or bad results [Bershad91].

Benchmarking issues become even more complex when studying OS/architecture interactions. Unlike experiments that only measure an application's interaction with the hardware, OS/architecture studies must consider how a workload exercises the hardware, the operating system, various software services, the interactions among these components,

1. The question of what is an appropriate workload has been widely debated by the research community for many years. The only conclusion is that there is no such thing as one appropriate set of workloads for all users. However, many studies have relied on a narrow range of workloads, neglecting to consider important software trends and their impact on design trade-offs.

2. Two reasons why researchers have been unable to consider a wider range of applications are the lack of application source code and tools capable of capturing the behavior of those applications.

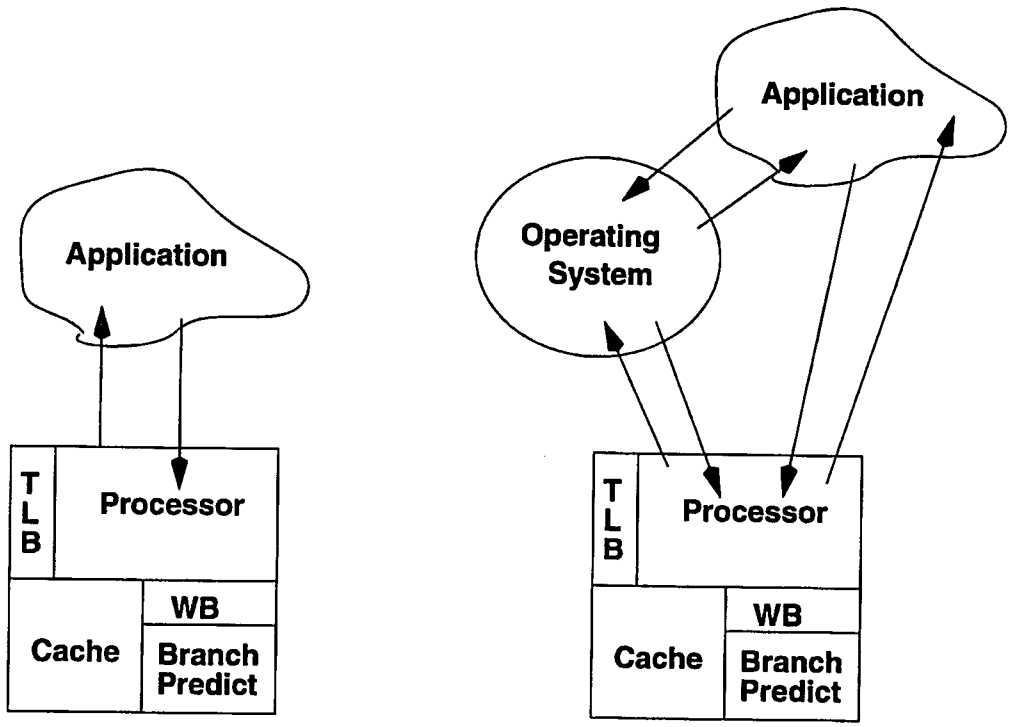


Figure 2.1 Complexity of OS/architecture studies

While traditional application/architecture studies consider only architecture and application interaction, OS/architecture studies must consider interactions among the operating system, application and architecture.

---

and the effects of competing workloads (Figure 2.3, p. 15). For example, decomposition of OS services can increase the amount of interprocess communication (IPC). It also can stress hardware components such as the TLB, but only when a significant number of services are concurrently used. Unfortunately, most OS-oriented benchmarks do not utilize a sufficient number of concurrent services, completely missing the interaction among OS, application and TLB (for a more detailed analysis, refer to Chapter 3).

Environmental issues must also be considered when measuring a system's performance. The number of tasks running, the task page allocation policy, and the size of physical memory play an important role in determining the experimental results. This is completely different from an application/architecture study, where the behavior of the

system is invariant between program runs. Our own measurements have shown that as Mach 3.0 ages, OS data structures can become scattered between the virtual and physical space, increasing the TLB miss rate. However, this type of phenomenon can only be observed after the system has been running for several hours or even several days.

Because these issues can have a pronounced effect on the results of an experiment, it is important that we examine how benchmarks should be used when studying OS/architecture interactions. The following sections summarize our findings. We begin by outlining basic benchmarking issues common to all types of performance research. This is followed by a discussion of benchmarking issues that are important to OS/architecture research. We conclude with a discussion of our benchmarks, why they were chosen, and some of their limitations.

2.3.1 Benchmarking 101

2.3.1.1 How benchmarks are used

Often, benchmarks are used to compare the speed or throughput of different systems. The speed measurement is like a race: each system runs the same set of benchmarks and the computer that finishes in the least amount of time is the winner. The throughput measurement is like a weightlifting contest. Each system runs a number of concurrent workloads and the one that can sustain the largest number of workloads per unit time is the winner. Benchmark suites such as SPEC Integer, SPEC Floating Point, SPEC SDM, SPEC SFS, and the Perfect Club are all designed to compare performance across various systems.

Benchmarks can also be used to determine the relative importance of various system components. For example, measurements of scientific applications show that they rely heavily on floating point computation and large data sets [Gee93] while measurements of video applications show a need for special integer computation.

Understanding the requirements of a user's workload allows designers to allocate system resources to deliver the best performance.

Finally, benchmarks are useful for studying the interactions between hardware and software components and for analyzing the impact of design trade-offs. Hardware structures such as caches or register windows have been implemented because workload studies showed that software could utilize these structures to improve performance.

2.3.1.2 Types of benchmarks

Hennessy and Patterson outline four basic types of benchmarks [Hennessy91].

- Programs: applications that are used by computer users. The SPEC Integer and Floating Point Benchmark Suites consist of real programs.
- Kernels: key fragments of code. The Livermore Loops are kernel benchmarks.
- Toy: real, but small programs such as quicksort or the 8-Queens game. Stanford University has a collection of small benchmarks, called the Small Benchmark Suite.
- Synthetic: programs designed to simulate some aspect of a workload's behavior, but not based on any real code. Whetstone and Drystone are synthetic benchmarks.

Any of these benchmark categories can be used to compare the performance of different systems. However, synthetic and toy benchmarks tend to be poor indicators of overall performance. Kernel benchmarks, also known as micro-benchmarks, are drawn from key fragments of code such as the computational loops in a matrix multiply routine or a system call to an OS service. These benchmarks can determine how well a machine performs a specific set of software operations. However, their narrow focus prevents kernels from providing information on the relative importance of various operations.

If well chosen, real program benchmarks can be the most useful because they provide a complete view of how the software utilizes the hardware. This allows designers to understand the relative importance of various hardware and software structures and to focus their attention on optimizations that will provide the greatest improvements in overall performance.

2.3.1.3 Evaluating the performance of a system

There are two basic methods for evaluating the performance of a system. The first compares the performance of two or more machines. Metrics such as benchmark execution time, SPEC marks, or throughput are used to make the comparisons, providing quantitative comparisons that allows one to determine which machine is better for his or her set of workloads. The second method is to evaluate a system's performance against a bound. Here, measurements such as miss rate, CPI and the number of instructions executed are used to determine how close to "optimal" the performance is. This type of measurement is typically used by designers who try to find inefficiencies in various system structures.

2.3.2 Using benchmarks to study OS/architecture interactions

2.3.2.1 What is a good set of benchmarks?

The answer to this question depends very heavily on the types of questions one is trying to answer. For example, a study focusing on the performance of file system design trade-offs would probably use workloads that exercised the file system. The benchmarks might be real programs such as compress and tar, or micro-benchmarks such as the Modified Andrew Benchmark [Ousterhout91]. In contrast, a study that is trying to determine the relative importance of the file system should use a wider range of benchmarks representative of the types of workloads a system would normally run.

For OS/architecture studies, both real applications and micro-benchmarks should be used. General performance trends should be measured using a large set of real applications. This prevents certain poorly performing components from dominating the results unless the components are critical to overall performance. Further, real applications determine the relative importance of various system components. Therefore, it is important to come as close as possible to real workloads when determining which hardware and software components are important to optimize. Once performance-critical hardware and software structures have been identified, micro-benchmarks can be used to examine the performance and interaction of specific components.

Neither type of benchmark is ideal. Micro-benchmarks suffer from several problems. First, they provide no understanding of the relative importance of various system components. This problem is best overcome by using representative applications to determine which hardware and/or software structures are important and then selecting micro-benchmarks that focus on the important structures. Second, micro-benchmarks can under or over emphasize certain hardware or software effects, such as cache misses, TLB faults and context switches. These changes in behavior can generate overly optimistic results or obscure the real cause of the performance problem.

Real applications do not suffer these drawbacks. They can, however, create very complex interactions among the hardware, the operating systems and system software services. While this is exactly the behavior a designer should measure, the complexity of the system can be overwhelming, making it difficult to isolate the source of specific performance problems.

2.3.2.2 The system environment

Another important consideration is the system environment. Unlike application/architecture studies where the behavior of the software does not change from run to run, OS behavior can dramatically change between program runs. Policies such as

virtual to physical page mapping and scheduling, non-deterministic events such as interrupts, the locations of files in either the file cache or on disk, and the current state of OS data structures all influence the behavior of the system at a particular point in time.

Of course, the important question is, "Do changes in OS behavior significantly impact results?" The answer is most definitely yes. Sites, Kessler and Uhlig [Sites88, Kessler91, Uhlig95] have shown that page mapping within an address space can increase the cache miss rate enough to increase run time by several percent. It is also possible that random page mappings between address spaces will increase miss rates during some program runs. We have also observed that aging in kernel data structures can decrease performance. For example, the Mach 3.0 memory allocation algorithm for kernel data structures does not always distinguish between mapped and unmapped memory. As the system ages, this results in a higher reliance on mapped kernel memory. A workload executing on a system that has been alive for several days will incur up to 400% more TLB misses than the same workload run on a freshly booted machine.

These types of behavior have a number of ramifications. Studies that collect only one trace and then use the trace to drive many simulations may be relying on a trace that is particularly good or bad. This might cause the work to overlook important issues or focus attention on issues that occur very infrequently. For systems that process the trace on-the-fly, the variability between program runs may obscure the benefits of different implementations, making it difficult to evaluate design trade-offs where performance gains are only expected to be a few percent.

2.3.2.3 Summary

Using benchmarks to study OS/architecture interactions is fundamentally different from benchmarking application/hardware interactions. Benchmarks must not only stress application/architecture interactions, but also OS/architecture interactions and application/OS interactions. This can create a very complex set of interactions, making it

difficult to isolate the cause(s) of performance problems. The irreproducable nature of the OS and environmental issues such as the system aging can cloud results, making system analysis even more difficult. But, all of these issues should be considered because they are all components of real system behavior.

2.4 Tools and techniques for analyzing OS/architecture interactions

There are a number of tools and techniques that can be used to analyze OS/architecture interactions. This section surveys these tools and how we have used them in our work. It is followed by a discussion of our monitoring system, Monster, and our simulation techniques.

2.4.1 Coarse-grain tools and measurement techniques

2.4.1.1 Time and ps

Two of the most useful tools for evaluating the high-level behavior of a system are the Unix commands time and ps. time reports the estimated number of seconds a task spends in user mode, kernel mode, and the amount of idle time. ps reports the amount of time all living tasks have spent in user and system mode. By executing the ps command at the beginning and end of a benchmark, we are able to determine how much time a benchmark spends utilizing system services, such as the X display server, the AFS or NFS daemons, or the Mach BSD Unix server. We also use this information to verify that our Monster traces contain roughly the same proportion of benchmark and system service execution times as reported by ps.

Time and ps data are generated by sampling the state of the system at specific time intervals. This can create some error. For example, a benchmark that incurs a significant number of TLB misses will spend a lot of time in the kernel handling the TLB misses.

However, on our systems, some of the TLB handling code cannot be interrupted. While this code is executing, the system will not sample its state for time and ps statistics. Therefore, time and ps will underestimate the amount of time a task spends executing in kernel mode.

2.4.1.2 Other system utilities: vmstat, pstat

The operating system keeps numerous statistics on its behavior and interaction with specific hardware components. Tools like vmstat and pstat provide information on virtual and physical memory requirements, I/O activity, paging and swapping, and the number of interrupts per second. We use this information to determine if the system's paging activity is dominating performance.

2.4.1.3 Profiling the kernel

Operating systems can be profiled to measure the number of times and the amount of time spent in each OS function. OS designers use this information to identify which areas of the OS are most heavily used and to focus optimization techniques on those areas. For example, McRae [McRae91] has shown that the interface between the machine dependent and machine-independent virtual memory functions in Mach is one source of performance loss. Optimizing this interface could significantly reduce the cost of virtual memory management in Mach.

Profiling can also be used to determine the cost of primitive OS functions such as flushing the cache or memory-to-memory copy (bcopy). Architects can use this information to redesign specific aspects of the architecture, reducing the frequency and/or cost of expensive primitives.

2.4.1.4 Software event counters

Software event counters embedded in the OS can be a very effective tool for measuring various performance problems. Usually incurring very low overhead (e.g., 3 or 4 instructions), counters can identify frequently used modules or objects. They can also identify a significant increase in the use of a module or object. This information can be invaluable to a designer tweaking a system's performance.

Software event counters, however, provide no information about the cost[1] of using a module or object. Therefore, one must be very careful when using counters to determine the importance of components because without knowing the cost, software counts can be misleading.

2.4.1.5 Summary of coarse-grained tools

Understanding the system's behavior at the highest level is an important part of measuring OS/architecture interactions. Tools such as time, ps, and vmstat help to characterize a workload's interactions with various software and hardware components. The major limitation of coarse-grained tools is their inability to provide detailed, cycle-by-cycle information about how the software is interacting with the hardware. This information is crucial to understanding the performance and behavior of the architecture and its interaction with the software system.

2.4.2 Fine-grained tools and measurement techniques

2.4.2.1 Cycle and event counters

To measure and understand the behavior of the hardware, it is important to obtain fine-grained, cycle-by-cycle measurements. One approach is to use hardware and software

1. Cost as measured by the number of cycles.

counters to time and count specific events. One of the basic measurements this technique can obtain is the hardware efficiency measure, CPI.

$$CPI = \text{number of cycles} / \text{number of instructions}$$

Using a cycle and an instruction counter that increments every instruction, we computed the CPI for each of our benchmarks (see Chapter 1).

Cycle counters can also measure the cost of software functions. By recording a cycle counter's value at the entry and exit points of a code fragment, we can build a histogram of the code fragment's execution times, allowing us to compute the average execution time and the distribution of execution times. This method provides a much more accurate picture than the micro-benchmark technique, because the timing can be collected while real workloads execute, allowing the measurements to accurately capture the effects of hardware stalls such as TLB and cache misses.

More sophisticated counters can record hardware events such as cache and TLB misses, write buffer stalls, mis-predicted branches, floating point and integer muliply/divide stalls and superscalar issue rates[1]. Currently, several microprocessors, including the DEC Alpha and IBM RS/6000, provide these types of on-chip counters. A recent work by Cvetanovic et al. [Cvetanovic94] used the Alpha's hardware monitoring counters to measure its performance across a range of workloads.

Counters are very good at obtaining information about the performance of hardware and software structures. However, they cannot be used to evaluate the effectiveness of design trade-offs. For this, we must turn to tracing tools used in conjunction with trace-driven simulators.

---

1. For superscalar processors.

2.4.2.2 Tracing tools

The most common example of software-based tracing tools are code annotation tools such as pixie and AE [Larus90, Smith91]. These tools annotate a program with extra instructions which generate a trace of all memory references. Recently, Chen [Chen93a] has extended the technique to include the operating system. By annotating every application, daemon, server and the operating system, Chen has been able to produce address traces which contain most of the system's activity. This is an important extension to the code annotation technique because it enables OS/architecture studies without additional hardware. Other researchers can use Chen's tools to annotate their operating systems and investigate OS/architecture interactions.

However, this approach suffers from several drawbacks. First, it is not possible to annotate the entire kernel, leaving some, perhaps important, portions of the operating system untraced. Second, it provides no information about the performance of the operating system on the hardware being traced. Hardware events such as cache misses and write buffer stalls are not reflected in the address trace. Detailed hardware models must be constructed in order to investigate performance problems. Third, the annotated code increases the run time of all software and changes the memory reference pattern. These changes can alter the behavior of the system and must be accounted for by the simulation tools.

A recently developed approach for tracing the operating system is called -driven simulation [Uhlig94c]. Using a hardware trap to expose events, such as cache or TLB misses, to the software, trap-driven simulation can be very fast because it relies on the hardware to process unimportant events such as cache hits. This reduces the time distortion caused by simulation. Further, because the hardware triggers the tracing mechanism, neither the application nor the operating system needs to be annotated. Hence,

there is almost no distortion in the memory system. We discuss Uhlig's prototype trap-driven simulator, Tapeworm, in Section 2.5.3 (p. 31).

A third type of tracing mechanism is a hardware monitoring system. Using a hardware monitor attached to the pins of the CPU, the monitor captures traces of all system activity, including address references and stall cycles. Because the monitor is autonomous from the processor, it can capture the traces without distorting the behavior of the system. However, when the hardware monitor's storage capacity fills, the processor must stall while the trace-buffer is processed. This creates distortion at regular intervals during a program run, using making outstanding asynchronous events such as disk I/Os appear almost instantaneous. We discuss our hybrid hardware/software monitoring system in Section 2.5.2 (p. 28).

2.4.2.3 Summary of tools and techniques

As we have seen, a wide range of tools and techniques can be used to help in the study of OS/architecture interaction. Coarse-grained tools should be used first, to understand issues such as 1) which services an application uses and how much time is spent in each service; 2) which kernel routines are used the most; 3) whether an application creates activity, such as paging, that will dominate all other performance issues. Fine-grained tools can then focus on specific areas of concern, measuring the cycle-by-cycle interactions between the hardware and software, and identifying hardware and software components that cause performance loss.

2.5 Our approach

The previous sections have outlined various issues that are important to consider when studying OS/architecture interactions. In this section, we present our approach and how it resolves some of the issues raised in the first part of this Chapter.

Workload	Description
gcc	The GNU C compiler (version 2.6)
groff	GNU C++ implementation of the UNIX nroff text formatting program (version 1.09).
gs	Ghostscript (version 2.4.1) distributed by the Free Software Foundation. Renders and displays a single postscript page with text and graphics in an X display window
IOzone	A sequential file I/O benchmark that writes and then reads a 10 Megabyte file. Written by Bill Norcott.
jpeg_play	The xloadimage program written by Jim Frost. Displays four JPG images.
mab	John Ousterhout's Modified Andrew Benchmark [Ousterhout89].
mpeg_play	mpeg_play V2.0 from the Berkeley Plateau Research roup. Displays 610 frames from a compressed video file [Patel92].
nroff	Unix text formatting program shipped with Ultrix 3.1.
ousterhout	John Ousterhout's benchmarks suite from [Ousterhout89].
sdet	A multiprocess, system performance benchmark which includes programs that test CPU, OS and I/O performance. From the SPEC SDM benchmark suite.
verilog	Verilog-XL (version 1.6b) simulating the logic design of an experimental microprocessor.
video_play	A modified version of mpeg_play that displays 610 frames of an uncompressed video file.

Table 2.3 Workloads

These are the workloads used throughout this dissertation. Some experiments use a subset of the entire suite or a single workload for illustration.

2.5.1 Workloads

As discussed in Section 2.5.1 (p. 27), the choice of workloads can completely determine the outcome of an experiment. For example, to show bad TLB performance one should use the SPEC nasa7 program or the UNIX compress utility; to show bad D-cache performance, use Spice or compress. However, our goal is to understand OS/architecture interactions across a range of applications and to put various performance trends into perspective. Therefore, we used a wide range of applications described in Table 2.3 (p. 27). We also describe the operating systems in Table 2.4 (p. 28).

Operating System	Description
Ultrix	Version 3.1 from Digital Equipment Corporation
OSF/1	OSF/1 1.0 is the Open Software Foundation's version of Mach 2.5
Mach 3.0	Carnegie Mellon University's version mk77 of the kernel and nk38 of the UNIX server
Mach3+AFSin	Same as Mach 3.0, but with the AFS cache manager (CM) running in the UNIX server
Mach3+AFSout	Same as Mach 3.0, but with the AFS cache manager running as a separate task outside of the UNIX server.

Table 2.4 Operating systems

The basic operating systems were used in this dissertation: Ultrix, OSF/1 and Mach 3.0. In Chapter 3, we use all 3 plus to variants on Mach to explore TLB performance issues. In Chapter 4, we only use Ultrix and Mach 3.0.

Workloads like mab and ousterhout exercise specific aspects of the OS/architecture interface. jpeg_play is CPU intensive. Most of the rest of the benchmarks are more general, representative of workloads like digital-media, simulation, text formatting, and multiprogramming. These are some of the most interesting workloads, revealing complex interactions between various hardware and software components.

2.5.2 Monster - a hybrid hardware/software monitoring system

Monster is a hybrid hardware/software monitoring system which allows us to monitor a MIPS R2000 based DECstation 3100 (Figure 2.4, p. 29). Monster consists of: 1) a Tektronix DAS 9200 logic analyzer that is attached to the R2000's CPU pins and 2) custom software modules written to control the logic analyzer, its built-in state-machine and the traces captured by the logic analyzer. With access to the CPU pins, Monster can observe every processor event including floating point, write buffer and cache stalls. This allows Monster to accurately monitor the systems activity by either counting and timing specific events or by capturing traces of the hardware activity.

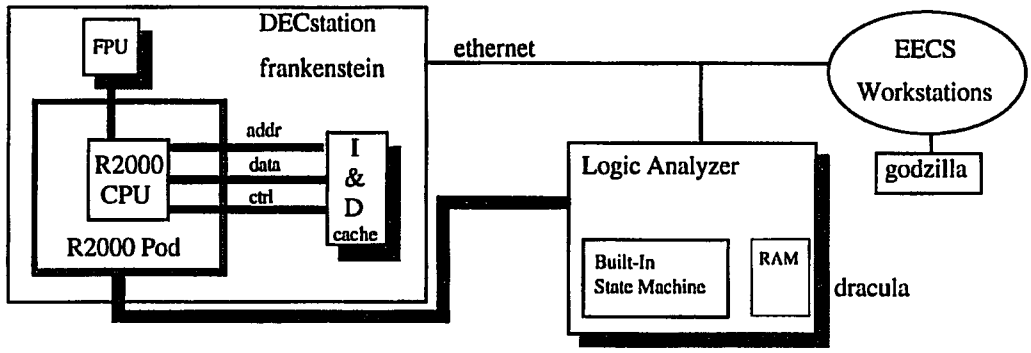


Figure 2.2 Monster monitoring system

Monster is attached to the CPU pins of a DECstation 3100. Because the caches are external to the R2000 processor, Monster can observe all system activity.

To count events, such as the number of instructions executed, we program the logic analyzer's built-in state machine and pattern matching hardware to watch for specific events. When the event occurs, a counter is incremented. This allows us to collect data measurements like CPI or miss rates without any distortion to the system under test. We also annotate the operating system with special markers[1] that denote specific points in the software. This allows us to locate and time events, such as kernel crossing or TLB misses. These markers do introduce some distortion into the system. However, we have measured the amount of distortion introduced and it is usually under 2%. Further, most of the experiments can correct for the distortion.

It is also possible to gather fine-grained timings by using the logic analyzer's memory buffer to record timestamps. We use this technique to measure the average cost of a TLB miss. The TLB miss handler is annotated with markers at its entry and exit points. Every time the logic analyzer detects one of the markers, it write the current timestamp into its buffer memory (Figure 2.5, p. 30). Once the buffer is full, the timestamps are

1. The markers are implemented by performing an uncached load into register R0 from an address in the kernel's text segment. This technique was taken from Torrellas [Torrellas92].

Marker	Time (in nanoseconds)	Delta
tlb miss entry	200	
tlb miss exit	1400	1200
tlb miss entry	20000	
tlb miss exit	21400	1400
tlb miss entry	42000	
tlb miss exit	43000	1000

Figure 2.3 Logic analyzer buffer with timestamps

The logic analyzer buffer can store timestamps for specific events detected by the state machine. Processing the data generates a histogram of timings that can be used to compute various statistics such as the average amount of time spent in a routine and the standard deviation.

---

processed and used to compute the average amount of time spent in each TLB miss routine.

2.5.2.1 Monster traces

Probably the most important use of the logic analyzer is to gather traces of the system's activity. The traces include all instruction and data references as well as every stall cycle. We use the traces to drive various trace-driven simulators that model and analyze various system performance issues.

Because of the finite amount of space available in Monster's trace buffer (512-K entries or 256-K cycles), we adopt two different approaches to gathering traces. The first approach forces the system to stall while the trace buffer is emptied (Figure 2.6, p. 31). This is done by entering a stall loop in the kernel's clock interrupt handler. Using this method, we can gather contiguous traces of each workload. The second approach, called trace sampling, does not stall the processor when the buffer fills. Instead, the workload is executed multiple times and during each program run, a sample is captured from a different portion of the workload. Capturing 50 or more samples usually provides enough data to accurately characterize specific aspects of the system's behavior.

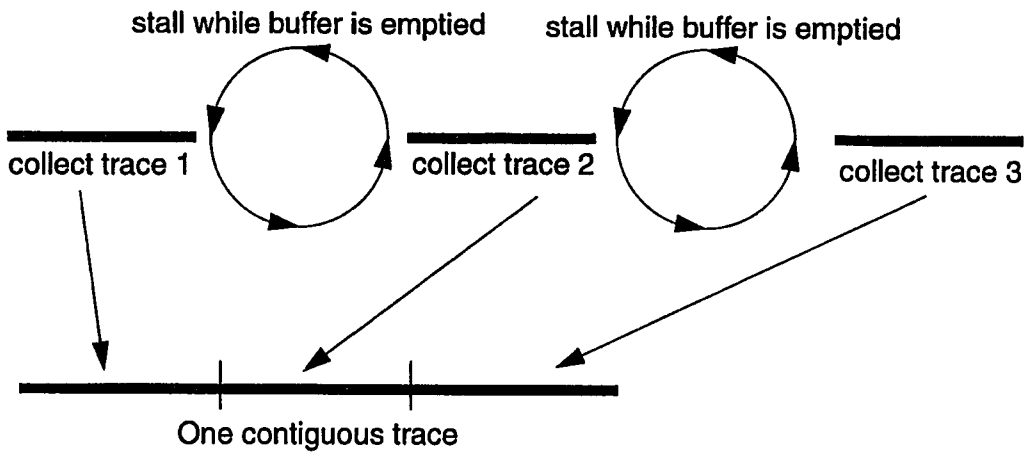


Figure 2.4 Capturing a contiguous trace

To capture a contiguous trace, the operating system enters a stall loop every time the logic analyzer buffer fills. Once the buffer is emptied, the operating system exits the loop and resumes execution.

The major drawback of collecting contiguous traces is that activity such as disk I/O can appear almost instantaneous. However, this does not affect the performance issues we examine. Trace sampling overcomes this problem, but each trace is very small, usually containing 100,000 instruction addresses and 30,000 data references, limiting their use in trace-driven simulation. However, because the segments are un-distorted, they are very good for analyzing the behavior of the system.

2.5.3 Tapeworm - a trap-driven simulation tool

Trace-driven simulation has become the de facto standard for evaluating hardware trade-offs. Relying on the fact that an application's trace is invariant to changes in the hardware structure, a single address trace can be used to explore many different types of architectural trade-offs.

However, once OS and multiprocess references are included in the address trace, the trace is no longer invariant to changes in the hardware. In fact, any change in the

hardware could alter the address stream, possibly creating very different interactions than those found in the original trace. This limits the use of trace-driven simulation when analyzing OS/architecture interactions.

To overcome this problem, Uhlig has developed a technique called trap-driven simulation [Uhlig94c]. The technique uses the real hardware to model a simulated architecture by setting traps on various data structures, such as a simulated TLB or cache. Whenever the real machine touches an "invalid" entry in one of these data structures, the hardware traps to the operating system where the trap-driven simulator is invoked. The simulator records the event and then validates the data structure entry to reflect the change in the simulated architecture's state, and then returns control to the executing application. This allows trap-driven simulation to accurately reflect variations in the address stream due to changes in the simulated architecture.

The original prototype trap-driven simulator, Tapeworm I, is capable of simulating various TLB designs (Figure 2.7, p. 33). Relying on the fact that the MIPS R2000 handles all TLB misses in software, calls to the Tapeworm simulator are placed at the beginning of all TLB miss handlers. On every TLB miss, the Tapeworm simulator checks to see if the real miss would have resulted in a miss or hit in the simulated TLB. If a simulated miss occurs, Tapeworm handles it and then returns to the real TLB handlers. For simulated TLBs larger than the MIPS R200 TLB (64 entries), Tapeworm maintains a simulated TLB data structure. For simulated TLBs smaller than on the R2000, Tapeworm modifies the OS TLB handlers so that the physical behavior is identical to the simulated TLB. For example, to model a 32 entry TLB, Tapeworm would instruct the OS TLB handlers to use only 32 entries of the real TLB. For a complete description of the Tapeworm and enhancements to the simulation technique, the reader is referred to [Uhlig95].

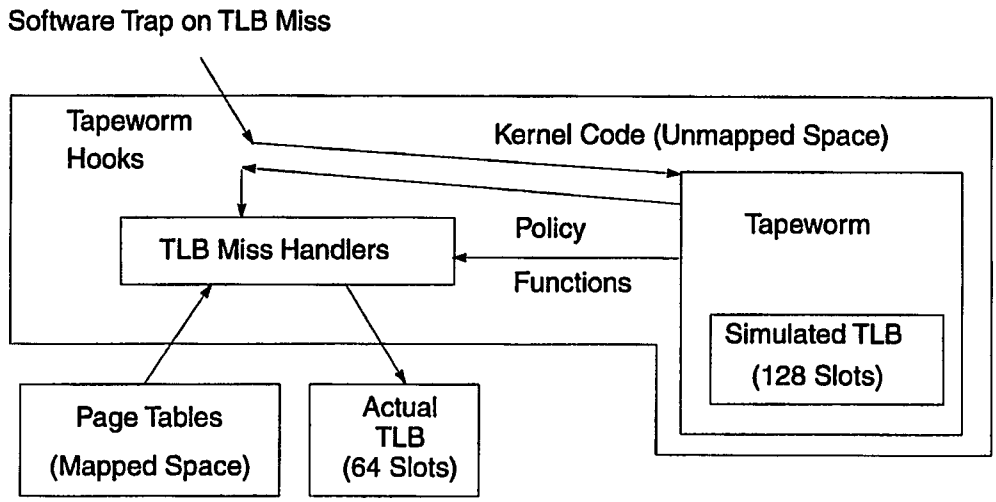


Figure 2.5 Tapeworm simulator

The Tapeworm TLB simulator is built into the operating system and is invoked whenever there is a real TLB miss. The simulator uses the real TLB misses to simulate its own TLB configuration(s). Because the simulator resides in the operating system, Tapeworm captures the dynamic nature of the system and avoids the problems associated with simulators driven by static traces.

2.5.4 Verifying our measurements and results

It was important to guarantee that the results accurately reflect the behavior of the system being measured. Three different measurements were used to verify the results. First, each workload's CPI is measured with Monster's instruction and cycle counters, which are very accurate. Second, at least one continuous trace of each workload is collected and analyzed to determine the source of stalls. Many times a sample-based trace is also collected and analyzed. Sampling does not stall the processor during sample collection. Instead, each sample is taken during a separate program run. This allows us to collect a set of 50 to 200 samples from various parts of a program and guarantees that tracing does not affect the program execution. Third, traces are run though a software simulation of a DECstation 3100. All of the measurements for each benchmark are then

compared. Typically, we see a small amount of variation, 3% - 5%, which is due to the dynamic behavior of the system. Any variation larger than 10% usually suggests a problem with the measurements and is investigated further.

2.5.5 Exploring architectural trade-offs

Throughout this dissertation, we rely on both Tapeworm and trace-driven simulation tools to model various architectural trade-offs. Tapeworm I was used in the TLB studies in Chapter 3. Trace-driven simulators driven by Monster traces were used to model the cache and prefetching simulations in Chapter 4. In each instance, the simulators were verified against real hardware measurements as discussed in Section 2.5.4.

2.6 Summary

The data in Chapter 1 provided a firm understanding of the OS/architecture performance problems. We now apply the tools and techniques described in this chapter to relate our performance data to the rest of the system. Our goal is to uncover the sources of the performance problems and use that information to help guide our exploration of the architectural design space.

CHAPTER 3

TLBs and their Interaction with the Operating System

3.1 OS impact on software-managed TLBs

Operating system references can have a strong influence on TLB performance. Yet, few studies have examined these effects, with most confined to a single operating system [Clark85, DeMoney86]. However, differences between operating systems can be substantial. To illustrate this point, we ran a set of benchmarks on each of the operating systems listed in Table 2.4 (p. 28). The results (Table 3.1) show that although the same application binaries were run on each system, there is significant variance in the number of TLB misses and total TLB service time. Some of these increases are due to differences in

Operating System	Run Time (secs)	Total Number of TLB Misses	Total TLB Service Time (secs)*	Ratio to Ultrix TLB Service Time
Ultrix 3.1	583	9,177,401	11.82	1.0
OSF/1	892	11,691,398	51.85	4.39
Mach 3.0	975	24,349,121	80.01	6.77
Mach 3+AFSin	1,371	33,933,433	106.56	9.02
Mach3+AFSout	1,517	36,649,834	134.71	11.40

Table 3.1 Total TLB misses across the benchmarks

The total run-time and number of TLB misses incurred by the benchmarks compress, IOzone, jpeg_play, mab, mpeg_play, ousterhout and video_play. Although the same application binaries were run on each of the operating systems, there is a substantial difference in the number of TLB misses and their corresponding service times.

*Time based on measured median time to service TLB miss.

the functionality between operating systems (i.e. UFS vs. AFS). Other increases are due to the structure of the operating systems. For example, the monolithic Ultrix spends only 11.82 seconds handling TLB misses while the microkernel-based Mach 3.0 spends 80.01 seconds.

Notice that while the total number of TLB misses increases 4 fold (from 9,177,401 to 36,639,834 for AFSout), the total time spent servicing TLB misses increases by a a factor of 11.4. This is due to the fact that software-managed TLB misses fall into different categories, each with its own associated cost. For this reason, it is important to understand page table structure, its relationship to TLB miss handling and the frequencies and costs of different types of misses.

3.2 Page tables and translation hardware

OSF/1 and Mach 3.0 both implement a linear page table structure (Figure 3.1). Each task has its own level 1 (L1) page table, which is maintained by machine-independent pmap code [Rashid88]. Because the user page tables can require several megabytes of space, they are themselves stored in the virtual address space. This is supported through level 2 (L2 or kernel) page tables, which also map other kernel data. Because kernel data is relatively large and sparse, the L2 page tables are also mapped. This gives rise to a 3-level page table hierarchy and four different page table entry (PTE) types.

The R2000 processor contains a 64-slot, fully-associative TLB, which is used to cache recently-used PTEs. When the R2000 translates a virtual address to a physical address, the relevant PTE must be held by the TLB. If the PTE is absent, the hardware invokes a trap to a software TLB miss handling routine that finds and inserts the missing PTE into the TLB. The R2000 supports two different types of TLB miss vectors. The first, called the user TLB (uTLB) vector, is used to trap on missing translations for L1U pages. This vector is justified by the fact that TLB misses on L1U PTEs are typically the most frequent [DeMoney86]. All other TLB miss types (such as those caused by references to

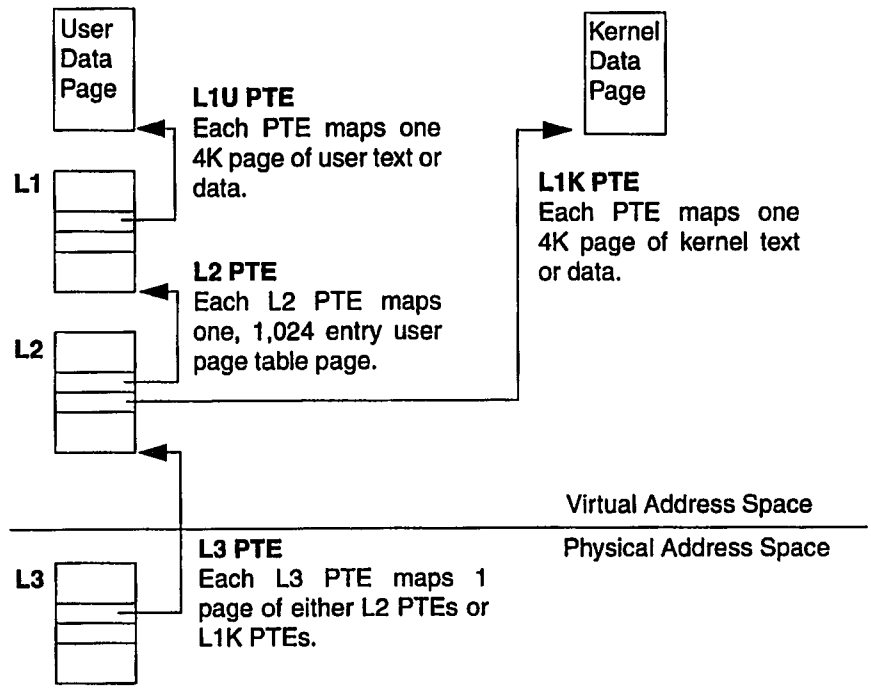


Figure 3.1 Page table structure in OSF/1 and Mach 3.0

The Mach page tables form a 3-level structure with the first two levels residing in virtual (mapped) space. The top of the page table structure holds the user pages which are mapped by level 1 user (L1U) PTEs. These L1U PTEs are stored in the L1 page table with each task having its own set of L1 page tables.

Mapping the L1 page tables are the level 2 (L2) PTEs. They are stored in the L2 page tables which hold both L2 PTEs and level 1 kernel (L1K) PTEs. In turn, the L2 pages are mapped by the level 3 (L3) PTEs stored in the L3 page table. At boot time, the L3 page table is fixed in unmapped physical memory. This serves as an anchor to the page table hierarchy because references to the L3 page table do not go through the TLB.

The MIPS R2000 architecture has a fixed 4 KByte page size. Each PTE requires 4 bytes of storage. Therefore, a single L1 page table page can hold 1,024 L1U PTEs, or 4 Megabytes of virtual address space. Likewise, the L2 page tables can directly map either 4 Megabytes of kernel data or indirectly map 4 GBytes of L1U data.

TLB Miss Type	Ultrix	OSF/1	Mach 3.0
L1U	16	20	20
L1K	333	355	294
L2	494	511	407
L3	—	354	286
Modify	375	436	499
Invalid	336	277	267

Table 3.2 Costs for different TLB miss types

This table shows the number of machine cycles (at 60 ns/cycle) required to service different types of TLB misses. To determine these costs, Monster was used to collect a 128K-entry histogram of timings for each type of miss. We separate TLB miss types into the six categories described below. Note that Ultrix does not have L3 misses because it implements a 2-level page table.

L1U TLB miss on a level 1 user PTE.

L1K TLB miss on a level 1 kernel PTE.

L2 TLB miss on level 2 PTE. This can only occur after a miss on a level 1 user PTE.

L3 TLB miss on a level 3 PTE. Can occur after either a level 2 miss or a level 1 kernel miss.

Modify A page protection violation.

Invalid An access to an page marked as invalid (page fault).

kernel pages, invalid pages or read-only pages) and all other interrupts and exceptions trap to a second vector, called the generic exception vector.

For the purposes of this study, we define TLB miss types (Table 3.2) to correspond to the page table structure implemented by OSF/1 and Mach 3.0. In addition to L1U TLB misses, we define five subcategories of kernel TLB misses (L1K, L2, L3, modify and invalid). Table 3.2 also shows our measurements of the time required to handle the different types of TLB misses. The wide differential in costs is primarily due to the two different miss vectors and the way that the OS uses them. L1U PTEs can be retrieved within 16 cycles because they are serviced by a highly-tuned handler inserted at the uTLB vector. However, all other miss types require from about 300 to over 400 cycles because they are serviced by the generic handler residing at the generic exception vector.

OS	Mapped Kernel Data Structures	Service Migration	Service Decomp.	Add. OS Services
Ultrix	Few	None	None	X Server
OSF/1	Many	None	None	X Server
Mach 3.0	Some	Some	Some	X Server
Mach3+AFSin	Some	Some	Some	X Server & AFS CM
Mach3+AFSout	Some	Some	Many	X Server & AFS CM

Table 3.3 Characteristics of the operating systems studied

The R2000 TLB hardware supports partitioning of the TLB into two sets of slots. The lower partition is intended for PTEs with high retrieval costs, while the upper partition is intended to hold more frequently-used PTEs that can be re-fetched quickly (e.g. L1U) or infrequently-referenced PTEs (e.g L3). The TLB hardware also supports random replacement of PTEs in the upper partition through a hardware index register that returns random numbers in the range 8 to 63. This effectively fixes the TLB partition at 8, so that the lower partition consists of slots 0 through 7, while the upper partition consists of slots 8 through 63.

3.3 OS influence on TLB performance

In the operating systems studied, there are three basic factors which account for the variation in the number of TLB misses and their associated costs (Table 3.2 & Figure 3.2). The central issues are (1) the use of mapped memory by the kernel (both for page tables and other kernel data structures), (2) the placement of functionality within the kernel, within a user-level server process (service migration) or divided among several server processes (OS decomposition) and (3) the range of functionality provided by the system (additional OS services). The rest of Section 3.1 uses our data to examine the relationship between these OS characteristics and TLB performance.

3.3.1 Mapping kernel data structures

Mapping kernel data structures adds a new category of TLB misses: L1K misses. In the MIPS R2000 architecture, an increase in the number of L1K misses can have a substantial impact on TLB performance because each L1K miss requires several hundred cycles to service[1].

Ultrix places most of its data structures in a small, fixed portion of unmapped memory that is reserved by the OS at boot time. However, to maintain flexibility, Ultrix can draw upon the much larger virtual space if it exhausts this fixed-size unmapped memory. Table 3.4 shows that few L1K misses occur under Ultrix.

In contrast, OSF/1 and Mach 3.0[2] place most of their kernel data structures in mapped virtual space, forcing them to rely heavily on the TLB. Both OSF/1 and Mach 3.0 mix the L1K PTEs and L1U PTEs in the TLB's 56 upper slots. This contention produces a large number of L1K misses. Further, handling an L1K miss can result in an L3 miss[3]. In our measurements, OSF/1 and Mach 3.0 both incur more than 1.5 million L1K misses. OSF/1 must spend 62% of its TLB handling time servicing these misses while Mach 3.0 spends 37% of its TLB handling time servicing L1K misses.

3.4 Service migration

In a traditional operating system kernel such as Ultrix or OSF/1 (Figure 3.2), all OS services reside within the kernel, with only the kernel's data structures mapped into the virtual space. Many of these services, however, can be moved into separate server tasks,

---

1. From 294 to 355 cycles, depending on the operating system (Table 3.3).

2. Like Ultrix, Mach 3.0 reserves a portion of unmapped space for dynamic allocation of data structures. However, it appears that Mach 3.0 quickly uses this unmapped space and must begin to allocate mapped memory. Once Mach 3.0 has allocated mapped space, it does not distinguish between mapped and unmapped space, despite their differing costs.

3. L1K PTEs are stored in the mapped L2 page tables (Figure 3.1).

System	Total Run Time (sec)	L1U	L1K	L2	L3	Invalid	Modify	Total
Ultrix	583	9,021	136	3.8	—	16.2	115.0	9,177
OSF/1	892	9,818	1,510	35.0	207.1	79.3	42.5	11,691
Mach3	975	21,466	1,683	352.7	556.3	165.9	125.4	24,349
Mach3+AFSin	1,371	30,123	2,493	330.8	690.4	168.4	127.2	33,933
Mach3+AFSOut	1,517	31,611	2,713	1,042.6	987.7	168.1	127.5	36,649

Table 3.4 Number of TLB misses (in thousands)

System	Total TLB Service Time (sec)	L1U	L1K	L2	L3	Invalid	Modify	% of Total Run Time
Ultrix	11.82	8.66	2.71	0.11	—	0.33	0.00	2.03%
OSF/1	51.85	11.78	32.16	1.07	4.40	1.32	1.11	5.81%
Mach3	80.01	25.76	29.68	8.61	9.55	2.66	3.75	8.21%
Mach3+AFSin	106.56	36.15	43.98	8.08	11.85	2.70	3.81	7.77%
Mach3+AFSOut	134.71	37.93	47.86	25.46	16.95	2.69	3.82	8.88%

Table 3.5 Time spent handling TLB misses (in seconds)

These tables show the number of TLB misses and amount of time spent handling TLB misses for each of the operating systems studied. In Ultrix, most of the TLB misses and TLB miss time is spent servicing L1U TLB misses. However, for OSF/1 and various versions of Mach 3.0, L1K and L2 misses can overshadow the L1U miss time. The increase in Modify misses is due to OSF/1 and Mach 3.0's use of protection to implement copy-on-write memory sharing.

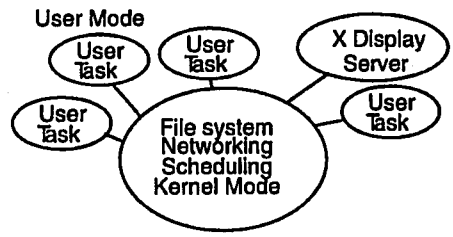
increasing the modularity and extensibility of the operating system [Anderson91]. For this reason, numerous microkernel-based operating systems have been developed in recent years (e.g. Chorus [Dean91], Mach 3.0 [Acetta86], V [Cheriton84]).

By migrating these services into separate user-level tasks, operating systems like Mach 3.0 fundamentally change the behavior of the system for two reasons. First, moving OS services into user space requires both their program text and data structures to be mapped. Therefore, they must share the TLB with user tasks, possibly conflicting with the user tasks' TLB footprints. Comparing the number of L1U misses in OSF/1 and Mach 3.0, we see a 2.2 fold increase from 9.8 million to 21.5 million. This is directly due to moving OS services into mapped user space. The second change comes from moving OS data structures from mapped kernel space to mapped user space. In user space, the data structures are mapped by L1U PTEs which are handled by the fast uTLB handler (20 cycles for Mach 3.0). In contrast, the same data structures in kernel space are mapped by L1K PTEs which are serviced by the general exception (294 cycles for Mach 3.0).

3.5 Operating system decomposition

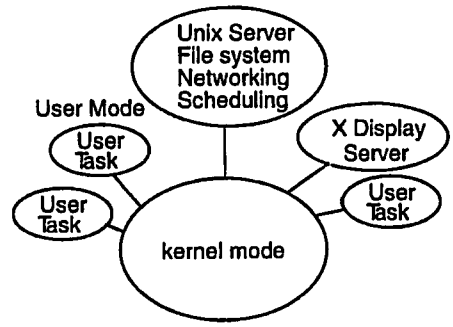
Moving OS functionality into a monolithic UNIX server does not achieve the full potential of a microkernel-based operating system. Operating system functionality can be further decomposed into individual server tasks. The resulting system is more flexible and can provide a higher degree of fault tolerance.

Unfortunately, experience with fully decomposed systems has shown severe performance problems. Anderson et al. [Anderson91] compared the performance of a monolithic Mach 2.5 and a microkernel Mach 3.0 operating system with a substantial portion of the file system functionality running as a separate AFS cache manager task. Their results demonstrate a significant performance gap between the two systems with Mach 2.5 running 36% faster than Mach 3.0, despite the fact that only a single additional



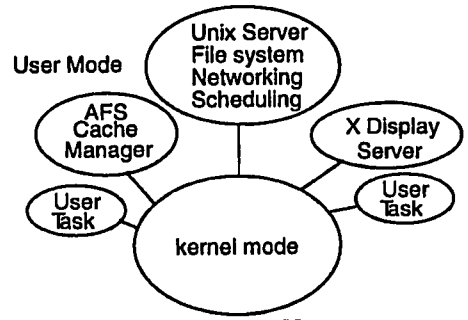
Ultrix and OSF/1

File system, networking, scheduling and Unix interface reside inside a monolithic kernel. Kernel text resides in unmapped space. Ultrix places most kernel data structures in unmapped space while OSF/1 uses mapped space for many of its kernel data structures.



Mach 3.0

File system, networking, and Unix interface reside inside the monolithic Unix Server. Kernel text and some data reside in unmapped virtual space but the Unix Server is in mapped user space.



Mach 3.0 + AFSout

Same as standard Mach 3.0, but with increased functionality provided by a server task. The AFS Cache Manager is either inside the Unix Server or its own, user-level server (as pictured above).

Figure 3.2 Monolithic and microkernel operating systems

A comparison of the monolithic Ultrix and OSF/1 and the microkernel Mach 3.0. In Ultrix and OSF/1, all OS services reside inside the kernel. In Mach 3.0, these services have been moved into the UNIX server. Therefore, most of Mach 3.0's functionality resides in mapped virtual space. Mach3+AFS is a modified version of Mach 3.0 with the AFS Cache Manager residing in either the Unix Server (AFSin) or as a separate user-level server (AFSout).

server task is used. Later versions of Mach 3.0 have overcome this performance gap by integrating the AFS cache manager into the UNIX Server.

We compared our benchmarks running on the Mach3+AFSin system, against the same benchmarks running on the Mach3+AFSout system. The only structural difference between the systems is the location of the AFS cache manager. The results (Table 3.5) show a substantial increase in the number of both L2 and L3 misses. Many of the L3 misses are due to missing mappings needed to service L2 misses.

The L2 PTEs compete for the R2000's 8 lower TLB slots. Yet, the number of slots required is proportional to the number of tasks concurrently providing an OS service. As a result, adding just a single, tightly-coupled service task overloads the TLB's ability to map L2 page tables. Thrashing results. This increase in L2 misses will grow ever more costly as systems continue to decompose services into separate tasks.

3.6 Additional OS functionality

In addition to OS decomposition and migration, many systems provide supplemental services (e.g. X, AFS, NFS, Quicktime). Each of these services, when interacting with an application, can change operating system behavior and how the operating system interacts with the TLB hardware.

For example, adding a distributed file service (in the form of an AFS cache manager) to the Mach 3.0 Unix server adds 10.39 seconds to the L1U TLB miss handling time (Table 3.5). This is due solely to the increased functionality residing in the Unix server. However, L1K misses also increase, adding 14.3 seconds. These misses are due to the additional management the Mach 3.0 kernel must provide for the AFS cache manager. Increased functionality will have an important impact on how architectures support operating systems and to what degree operating systems can increase and decompose functionality.

3.7 Improving TLB performance

In this section, we examine hardware-based techniques for improving TLB performance under the operating systems analyzed in the previous section. However, before suggesting changes, it is helpful to consider the motivations behind the design of the R2000 TLB.

The MIPS R2000 TLB design is based on two principal assumptions [DeMoney86]. First, L1U misses are assumed to be the most frequent (> 95%) of all TLB miss types. Second, all OS text and most of the OS data structures (with the exception of user page tables) are assumed to be unmapped. The R2000 TLB design reflects these assumptions by providing two types of TLB miss vectors: the fast uTLB vector and the much slower general exception vector (described in Section 3.2). These assumption are also reflected in the partitioning of the 64 TLB slots into two disjoint sets of 8 lower slots and 56 upper slots (also described previously). The 8 lower slots are intended to accommodate a traditional UNIX task (which requires at least three L2 PTEs) and UNIX kernel (2 PTEs for kernel data), with three L2 PTEs left for additional data segments [DeMoney86].

Our measurements (Table 3.4) demonstrate that these design choices make sense for a traditional UNIX operating system such as Ultrix. For Ultrix, L1U misses constitute 98.3% of all misses. The remaining miss types impose only a small penalty. However, these assumptions break down for the OSF/1- and Mach 3.0-based systems. In these systems, the non-L1U misses account for the majority of time spent handling TLB misses. Handling these misses substantially increases the cost of software-TLB management (Table 3.5).

The rest of this section proposes and explores four hardware-based improvements for software-managed TLBs. First, the cost of certain types of TLB misses can be reduced by modifying the TLB vector scheme. Second, the number of L2 misses can be reduced by

increasing the number of lower slots[1]. Third, the frequency of most types of TLB misses can be reduced if more total TLB slots are added to the architecture. Finally, we examine the trade-offs between TLB size and associativity.

Throughout these experiments, software policy issues do not change from those originally implemented in Mach 3.0. The PTE replacement policy is FIFO for the lower slots and Random for the upper slots. The PTE placement policy stores L2 PTEs in the lower slots and all other PTEs in the upper slots. The effectiveness of these and other software-based techniques are examined in a related work [Uhlig93].

3.8 Additional TLB miss vectors

The data in Table 3.4 show a significant increase in L1K misses for OSF/1 and Mach 3.0 when compared against Ultrix. This increase is due to both systems' reliance on dynamic allocation of mapped kernel memory. The R2000's TLB performance suffers, however, because L1K misses must be handled by the costly generic exception vector which requires 294 cycles (Mach 3.0).

To regain the lost TLB performance, the architecture could vector all L1K misses through the uTLB handler, as is done in the newer R4000 processor. Based on our timing and analysis of the TLB handlers, we estimate that vectoring the L1K misses through the uTLB handler would reduce the cost of L1K misses from 294 cycles (for Mach 3.0) to approximately 20 cycles.

An additional refinement would be to dedicate a separate TLB miss vector for L2 misses. We estimate that the L2 miss service time would decrease from 407 cycles (Mach 3.0) to under 40 cycles.

Table 3.6 (p. 47) shows the same data for Mach3+AFSin as Table 3.4 (p. 41), but recomputed with the new cost estimates resulting from the refinements above. The result of combining these two modifications is that total TLB miss service time drops from

1. The newer MIPS R4000 processor [Kane92] implements both of these changes.

Type of PTE Miss	Counts	Previous Total Cost from Table 3.5 (secs)	New Total Cost (sec)	Time Saved (sec)
L1U	30,123,212	36.15	36.15	0.00
L2	330,803	8.08	0.79	7.29
L1K	2,493,283	43.98	2.99	40.99
L3	690,441	11.85	11.85	0.00
Modify	127,245	3.81	3.81	0.00
Invalid	168,429	2.70	2.70	0.00
Total	33,933,413	106.56	58.29	48.28

Table 3.6 Recomputed cost of TLB misses given additional miss vectors (Mach 3.0+AFSin)

Supplying a separate interrupt vector for L2 misses and allowing the uTLB handler to service L1K misses reduces their cost to 40 and 20 cycles, respectively. Their contribution to TLB miss time drops from 8.08 and 43.98 seconds down to 0.79 and 2.99 seconds, respectively.

106.56 seconds down to 58.29 seconds. L1K service time drops 93% and L2 miss service time drops 90%. More importantly, the L1K and L2 misses no longer contribute substantially to overall TLB service time. This minor design modification enables the TLB to much more effectively support a microkernel-style operating system with multiple servers in separate address spaces.

Multiple TLB miss vectors provide additional benefits. In the generic trap handler, dozens of load and store instructions are used to save and restore a task's context. Many of these loads and stores cause cache misses which require the processor to stall. As processor speeds continue to outstrip memory access times, the CPI in this save/restore region will grow, increasing the number of wasted cycles and making non-uTLB misses much more expensive. TLB-specific miss handlers should not suffer the same performance problems because they contain only a single data reference to load the missed PTE from the memory-resident page tables.

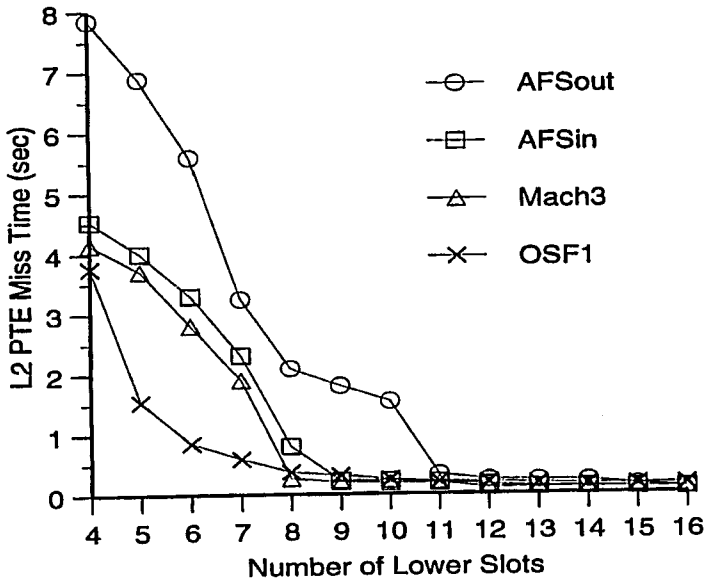


Figure 3.3 L2 PTE miss cost vs. number of lower slots

The total L2 miss time for the mab benchmark under different operating systems. As the TLB reserves more lower slots for L2 PTEs, the total time spent servicing L2 misses becomes negligible.

3.9 Lower slots & partitioning the TLB

The MIPS R2000 TLB fixes the partition between the 8 lower slots and the 56 upper slots. This partitioning is appropriate for an operating system like Ultrix [DeMoney86]. However, as OS designs migrate and decompose functionality into separate user-space tasks, having only 8 lower slots becomes insufficient. This is because, in a decomposed system, the OS services that reside in different user-level tasks compete by displacing each other's L2 PTE mappings from the TLB.

To better understand this effect, we measured how L2 miss rates vary depending on the number of lower TLB slots available. Tapeworm was used to vary the number of lower TLB slots from 4 to 16 while keeping the total number of TLB slots fixed at 64.

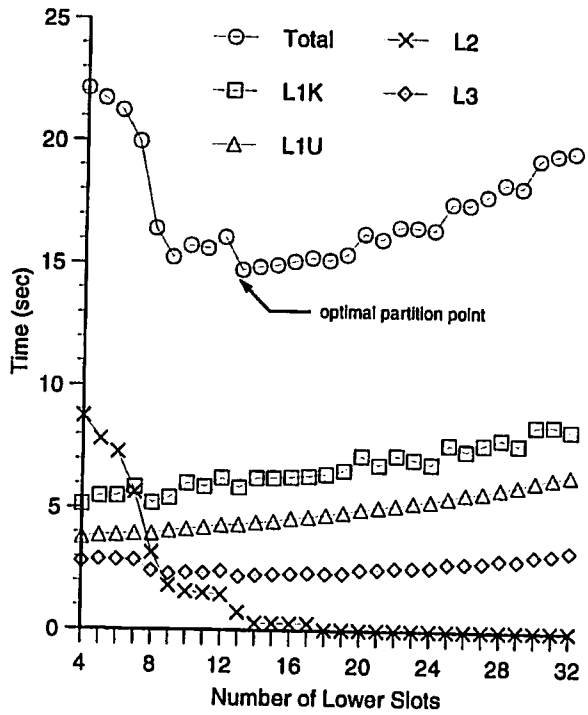


Figure 3.4 Total cost of TLB misses vs. number of lower TLB slots

The total cost of TLB miss servicing is plotted against the L1U, L1K, L2 and L3 components of this total time. The number of lower TLB slots varies from 4 to 32, while the total number of TLB entries remains constant at 64.

The benchmark is video_play running under Mach 3.0.

OSF/1 and all three versions of Mach 3.0 ran the mab benchmark over the range of configurations and the total number of L2 misses was recorded (Figure 3.3).

For each operating system, two distinct regions can be identified. The left region shows a steep decline which levels off near zero seconds. This shows a significant performance improvement for every extra lower TLB slot made available to the system, up to a certain point. For example, simply moving from 4 to 5 lower slots decreases OSF/1 L2 miss handling time by almost 50%. After 6 lower slots, the improvement slows because the TLB can hold most of the L2 PTEs required by OSF/1[1].

---

1. Two for kernel data structures and one each for a task's text, data and stack segments.

In contrast, the Mach 3.0 system continues to show significant improvement up to 8 lower slots. The additional 3 slots needed to bring Mach 3.0's performance in line with OSF/1 are due to the migration of OS services from the kernel to the UNIX Server in user space. In Mach 3.0, whenever a task makes a system call to the UNIX server, the task and the UNIX server must share the TLB's lower slots. In other words, the UNIX server's three L2 PTE's (text segment, data segment, stack segment) increases the lower slot requirement for the system as a whole to 8.

Mach3+AFSin's behavior is similar to Mach 3.0 because the additional AFS cache manager functionality is mapped by the UNIX server's L2 PTEs. However, when the AFS cache manager is decomposed into a separate user-level server, the TLB must hold three additional L2 PTEs (11 total). Figure 3.3 shows how Mach3+AFSout continues to improve until all 11 L2 PTEs can simultaneously reside in the TLB.

Unfortunately, increasing the size of the lower partition at the expense of the upper partition has the side-effect of increasing the number of L1U, L1K and L3 misses, as shown in Figure 3.4. Coupling the decreasing L2 misses with the increasing L1U, L1K and L3 misses yields an optimal partition point shown in Figure 3.4.

This partition point, however, is only optimal for the particular operating system. Different operating systems with varying degrees of service migration have different optimal partition points. For example, the upper graph in Figure 3.5 shows an optimal partition point of 8 for Mach 3.0, 10 for Mach3+AFSin and 12 for Mach3+AFSout, when running the Ousterhout benchmark.

Applications also influence the optimal partition point. The lower graph in Figure 3.5 (p. 51) shows the results for various applications running under Mach 3.0. compress has an optimal partition point of 8. However, video_play requires 14 slots and mpeg_play requires 18 slots. Some of the additional slots are used to hold the X Server's L2 PTEs. This underscores the importance of understanding both the decomposition of the

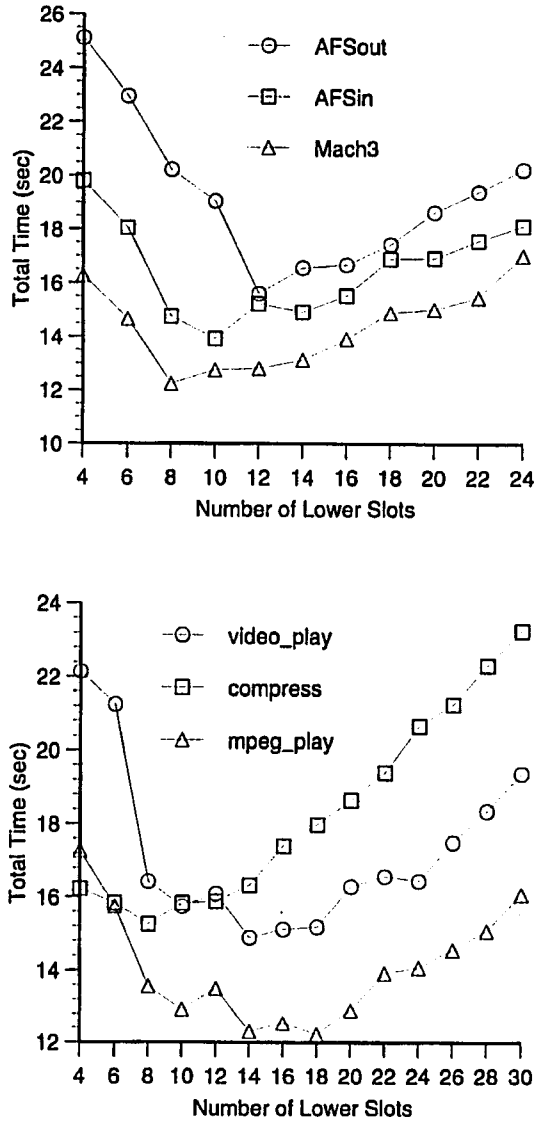


Figure 3.5 Optimal partition points for various operating systems and benchmarks

As more lower slots are allocated, fewer upper slots are available for the L1U, L1K and L3 PTEs. This yields an optimal partition point which varies with the operating system and benchmark.

The upper graph shows the average of 3 runs of the ousterhout benchmark run under 3 different operating systems. The lower graph shows the average of 3 runs for 3 different benchmarks run under Mach 3.0.

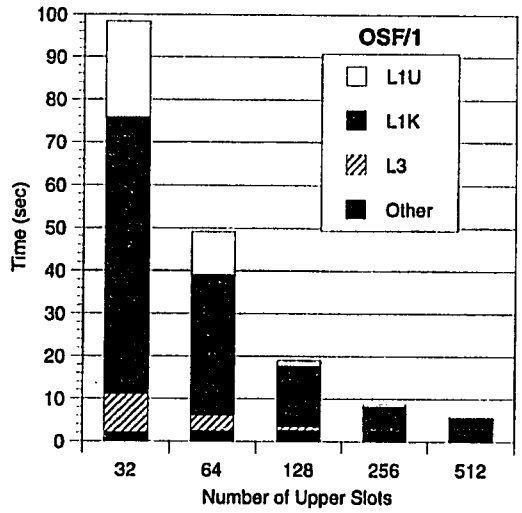


Figure 3.6 TLB service time vs. number of upper TLB slots

The total cost of TLB miss servicing for all seven benchmarks run under OSF/1. The number of upper slots was varied from 8 to 512, while the number of lower slots was fixed at 16 for all configurations.

system and how applications interact with the various OS services, because both determine the use of TLB slots.

3.10 Increasing TLB size

In this section we examine the benefits of building TLBs with additional upper slots. The trade-offs here can be more complex because the upper slots are used to hold three different types of mappings (L1U, L1K and L3 PTEs), whereas the lower slots only hold L2 PTEs.

To better understand the requirements for upper slots, we used Tapeworm to simulate TLB configurations ranging from 32 to 512 upper slots. Each of these TLB configurations was fully-associative and had 16 lower slots to minimize L2 misses.

Figure 3.6 shows TLB performance for all seven benchmarks under OSF/1. For smaller TLBs, the most significant component is L1K misses; L1U and L3 misses account for less than 35% of the total TLB miss handling time. The prominence of L1K misses is due to the large number of mapped data structures in the OSF/1 kernel. However, as outlined in Section 3.8, modifying the hardware trap mechanism to allow the uTLB handler to service L1K misses reduces the L1K service time to an estimated 20 cycles. Therefore, we recomputed the total time using the lower cost L1K miss service time (20 cycles) for the OSF/1, Mach 3.0 and Mach3+AFSout systems (Figure 3.7).

With the cost of L1K misses reduced, TLB miss handling time is dominated by L1U misses. In each system, there is a noticeable improvement in TLB service time as TLB sizes increase from 32 to 128 slots. For example, moving from 64 to 128 slots decreases Mach 3.0 TLB handling time by over 50%.

After 128 slots, invalid and modify misses dominate (listed as "other" in the figures). Because the invalid and modify misses are constant with respect to TLB size, any further increases in TLB size will have a negligible effect on overall TLB performance. This suggests that a 128- or 256-entry TLB may be sufficient to support both monolithic operating systems like Ultrix and OSF/1 and microkernel operating systems like Mach 3.0. Of course, even larger TLBs may be needed to support large applications such as CAD programs. However, this study is limited to TLB support for operating systems running a modest workload. The reader is referred to [Chen92] for a detailed discussion of TLB support for large applications.

3.11 TLB associativity

Large, fully-associative TLBs (128^{+} entries) are difficult to build[1] and can consume a significant amount of chip area. To achieve high TLB performance, computer

1. Current-mode sensing avoids some of the problems associated with large CMOS CAMs [Heald93].

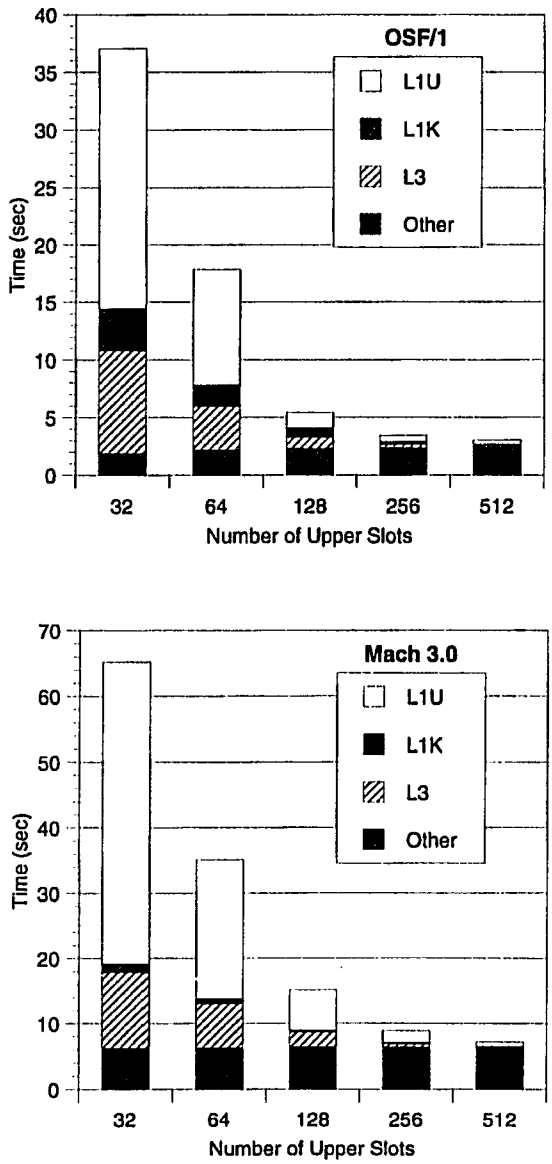


Figure 3.7 Modified TLB service time vs. number of upper TLB slots

The total cost of TLB miss servicing (for all seven benchmarks) assuming L1K misses can be handled by the uTLB handler in 20 cycles and L2 misses are handled in 40 cycles. The top graph is for OSF/1, the bottom for Mach 3.0. Note that the scale varies for each graph.

Other is the sum of the invalid, modify and L2 miss costs.

Processor	Associativity	Number of Instruction Slots	Number of Data Slots
DEC Alpha 21064	full	8+4	32
IBM RS/6000	2-way	32	128
TI Viking	full	64 unified	—
MIPS R2000	full	64 unified	—
MIPS R4000	full	48 unified	—
HP 9000 Series 700	full	96+4	96+4
Intel 486	4-way	32 unified	—

Table 3.7 Number of TLB slots for current processors

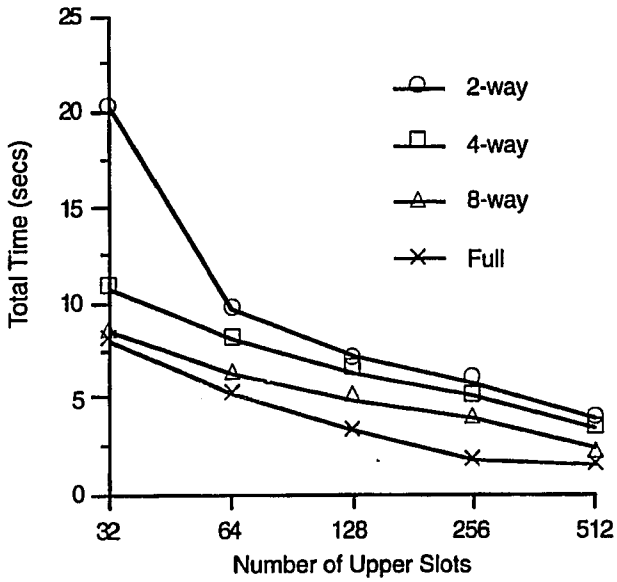
Note that page sizes vary from 4K to 16 MBytes and are variable in many processors. The MIPS R4000 actually has 48 double slots. Two PTEs can reside in one double slot if their virtual mappings are to consecutive pages in the virtual address space. [Talluri92]

architects could implement larger TLBs with lesser degrees of associativity. The following section explores the effectiveness of TLBs with varying degrees of associativity.

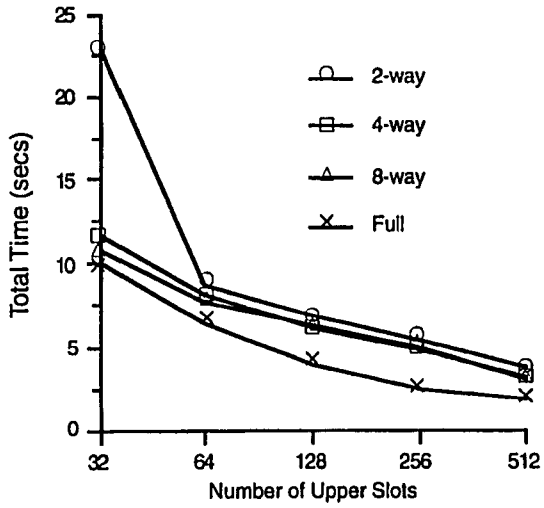
Many current-generation processors implement fully-associative TLBs with sizes ranging from 32 entries to 100^{+} entries (Table 3.7). However, technology limitations may force designers to begin building larger TLBs which are not fully-associative. To explore the performance impact of limiting TLB associativity, we used Tapeworm to simulate TLBs with varying degrees of associativity.

The graphs in Figure 3.9 (p. 57) show the total TLB miss handling time for the mpeg_play benchmark under Mach3+AFSout and the video_play benchmark under Mach 3.0. Throughout the range of TLB sizes, increasing associativity reduces the total TLB handling time. These figures illustrate the general "rule-of-thumb" that doubling the size of a caching structure will yield about the same performance as doubling the degree of associativity [Patterson90].

Some benchmarks, however, can perform badly for TLBs with a small degree of set associativity. For example, the graph in Figure 3.9 (p. 57) shows the total TLB miss handling time for the compress benchmark under OSF/1. For a 2-way set-associative TLB,



mpeg_play under Mach3_AFSout



video_play under Mach 3.0

Figure 3.8 Total TLB service time for TLBs of different sizes and associativities

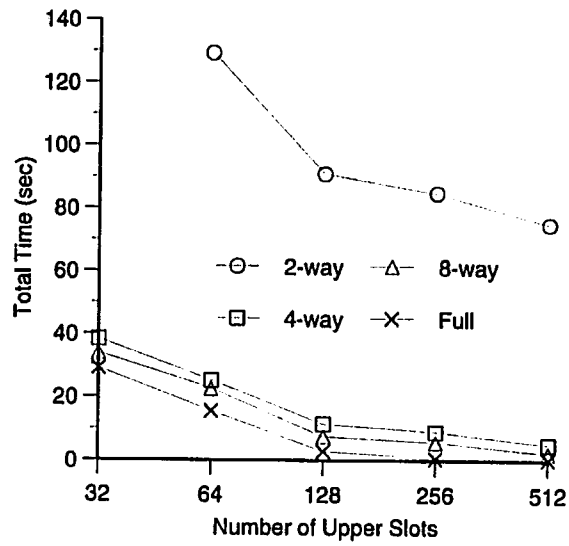


Figure 3.9 Total TLB service time for compress under OSF/1

compress displays pathological behavior. Even a 512-entry, 2-way set-associative TLB is outperformed by a much smaller 32-entry, 4-way set-associative TLB.

These three graphs show that reducing associativity to enable the construction of larger TLBs is an effective technique for reducing TLB misses.

3.12 Summary

This chapter demonstrates the importance of understanding the interactions between TLBs and operating systems. Software-management of TLBs magnifies the importance of this understanding, because of the large variation in TLB miss service times that can exist.

TLB behavior depends upon the kernel's use of virtual memory to map its own data structures, including the page tables themselves. TLB behavior is also dependent upon the division of service functionality between the kernel and separate user tasks. Currently popular microkernel approaches rely on server tasks, but can fall prey to

performance difficulties. Running on a machine with a software-managed TLB like that of the MIPS R2000, current microkernel systems perform poorly, even with only a modest degree of service decomposition into separate server tasks.

CHAPTER 4

Instruction Cache Performance and Design Trade-offs to Support Operating Systems

4.1 Introduction

It is well recognized that the continuing disparity between processor and memory speed continue to make instruction and data caches crucial to overall performance. The data in Figure 4.1 reaffirms this point, showing that both Ultrix and Mach spend approximately 70% of their stall cycles servicing cache misses.

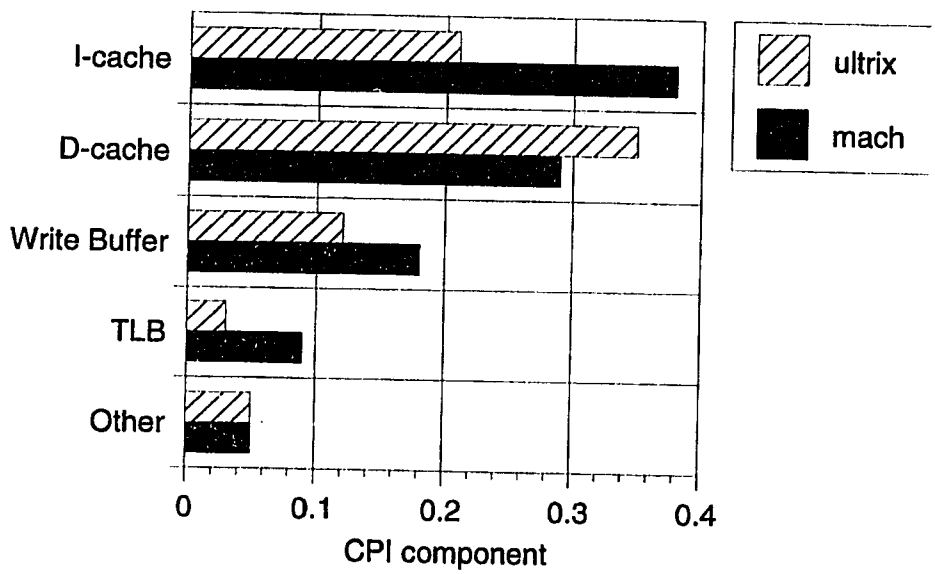


Figure 4.1 Stall breakdown by architectural component

The figure shows the architectural stall components for the workload suite* running on a DECstation 3100.

We excluded the micro-benchmarks IOzone, mab and ousterhout.

Figure 4.1 (p. 59) also shows a fundamental difference between Ultrix and Mach's cache behavior. While Ultrix stresses the D-cache more than I-cache, Mach's major source of stalls is the I-cache, over 60% greater than Ultrix. This was a surprising result because intuition suggested that the lack of locality in OS data structures should dominate cache performance. However, almost every benchmark consistently showed Mach's I-cache performance to be worse than its D-cache performance.

We believe that this shift, between D- and I-cache, represents a trend: that OS and software technologies are increasing the importance of the I-cache. For architects, this suggests a careful re-evaluation of I-cache designs, a difficult task since most commonly used benchmarks perform very well in very small I-caches [Gee94]. For OS designs, this performance problem could limit the advancement of OS technologies.

This chapter examines the reasons for and impact of increasing I-cache misses, using Monster traces to drive trace-driven simulators that model various memory system architectures. Section 4.2 analyzes how the structure of Mach increases the importance of the I-cache. Section 4.3 explores I-cache trade-offs and then uses those results in Section 4.4 to determine the best on-chip memory organization. The last section explores other architectural techniques, such as increasing bandwidth, prefetching, bypassing and instruction stream-buffers, that can further reduce stalls due to the instruction stream.

4.2 Reasons for increased I-cache misses

The longer invocation path to services in Mach is one of the major components responsible for the increase in I-cache misses. Our measurements[1] indicate that the round-trip call and return path to services in Ultrix (steps (a) and (b) in Figure 4.2 (p. 61) is less than 100 instructions. In contrast, the call path (1-4) under Mach consists of approximately 1000 instructions and the return path (5-7) uses about 850 instructions.

1. Similar measurements can be found in [Ford94].

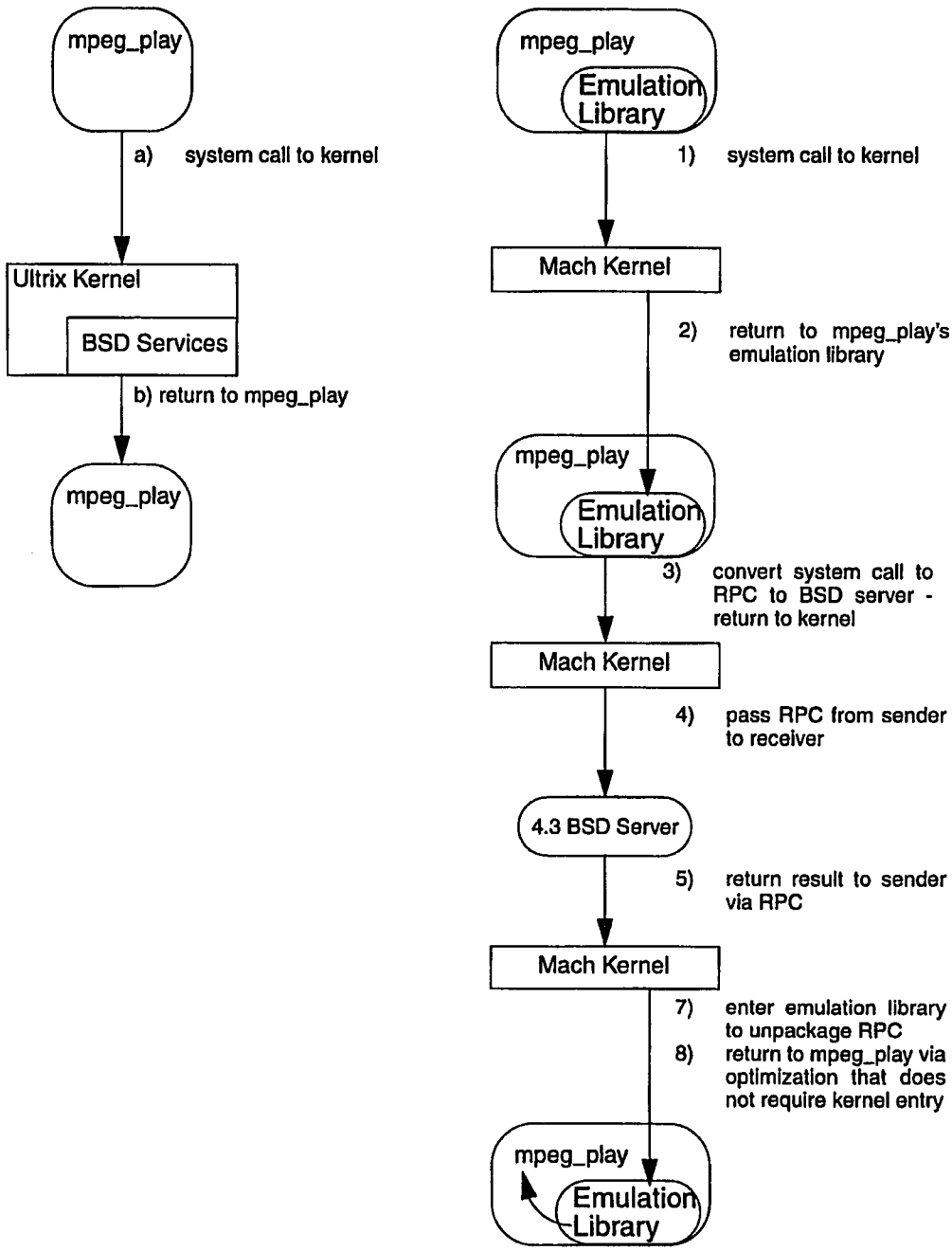


Figure 4.2 Service invocation paths in Ultrix and Mach

In Ultrix, BSD UNIX services reside in the kernel and are accessed through a single system call trap. In Mach, services reside in a user-level BSD server accessed via a remote procedure call (RPC) mechanism that passes through the Mach kernel.

Note that these are just service invocation paths. The actual OS code that provides the service must still execute in both Ultrix and Mach, but differences with respect to this service code are minor because both systems are derived from the same 4.2 BSD code [Chen93]. Using the instruction path lengths above, and assuming 4 bytes per instruction, we see that Mach increases the code path to OS services by approximately 4 K-bytes of instruction memory and the return path by about 3 K-bytes. These are long paths relative to the typical size of on-chip instruction caches in today's microprocessors. For example, a single system call under Mach will completely overrun a 4-Kbyte on-chip I-cache on the path to the BSD server, which will then have to warm the I-cache to run well. The return path overruns the I-cache as well, leaving the calling user task with a cold I-cache. Our measurements show that roughly one third of the time spent in the kernel during mpeg_play is due to the send and receive messages that compose RPCs, so this case happens frequently enough to have a substantial impact on overall I-cache performance.

Other structural differences are also responsible for increased I-cache misses under Mach. For example, in Ultrix, paging is implemented in the kernel, but Mach supports an external pager, running in user mode, that is responsible for locating pages in a backing store after a page fault [Draves91]. Similarly, recent versions of Mach have migrated I/O device drivers from the kernel to the user level [Forin91]. As a third example, Black et al. have described efforts to further decompose monolithic API servers (like the BSD server shown in Figure 4.2) into multiple, small-granularity servers (e.g., for naming, authentication, and file access) [Black92]. Each of these restructuring trends spreads out system code and further increases instruction path lengths between software modules.

It should be noted that the long path to system services under Mach is not a case of poor coding. This service invocation mechanism is common to other modular, object-oriented software systems (e.g. [Khalidi92]) and simply represents a cost for the advantages that they offer over traditional, single-service systems. In fact, the Mach implementation of RPC has been highly optimized through the use of techniques such as

stack-handoff scheduling and continuations [Draves91] for the common case of small messages, and out-of-line (virtual memory) transfers for the expensive case of large messages [Dean91].

Bershad has suggested other ways to avoid the costs of RPC, such as pre-allocating buffers between client and server address spaces for small messages using virtual memory primitives, or migrating more OS services into the client's address space as is currently done to a limited degree with the Mach emulation library [Bershad92]. Avoiding RPCs through more aggressive virtual memory sharing, however, is likely to shift misses from the I-cache to the TLB.

Though they have their performance penalties, these OS-structuring approaches offer important advantages. Dynamically-loaded emulation libraries enable binary compatibility. External pagers can simplify the implementation of distributed database or network-shared-memory applications. User-level device drivers are easier to debug, port and install, and small-granularity OS servers enhance opportunities for code reuse.

In summary, the functional advantages of modular, object-oriented software systems have been extensively documented in the literature. They include enhanced code re-usability, increased fault tolerance and ease of service distribution. We accept these trends as given, but note that they shift utilization of hardware components and thus prompt a re-evaluation of hardware architectures. In particular, the importance of I-caches and TLBs seems to increase relative to D-caches according to direct hardware measurements of a machine that implements its caches off-chip.

4.3 Instruction cache performance

The top left graph in Figure 4.3 (p. 64) shows that under Ultrix, small on-chip I-caches have fairly low miss ratios. For example, the miss ratio for an 8K-byte I-cache with a 4-word (16-byte) line is 0.028 and a 32K-byte I-cache with a 4-word line size has a

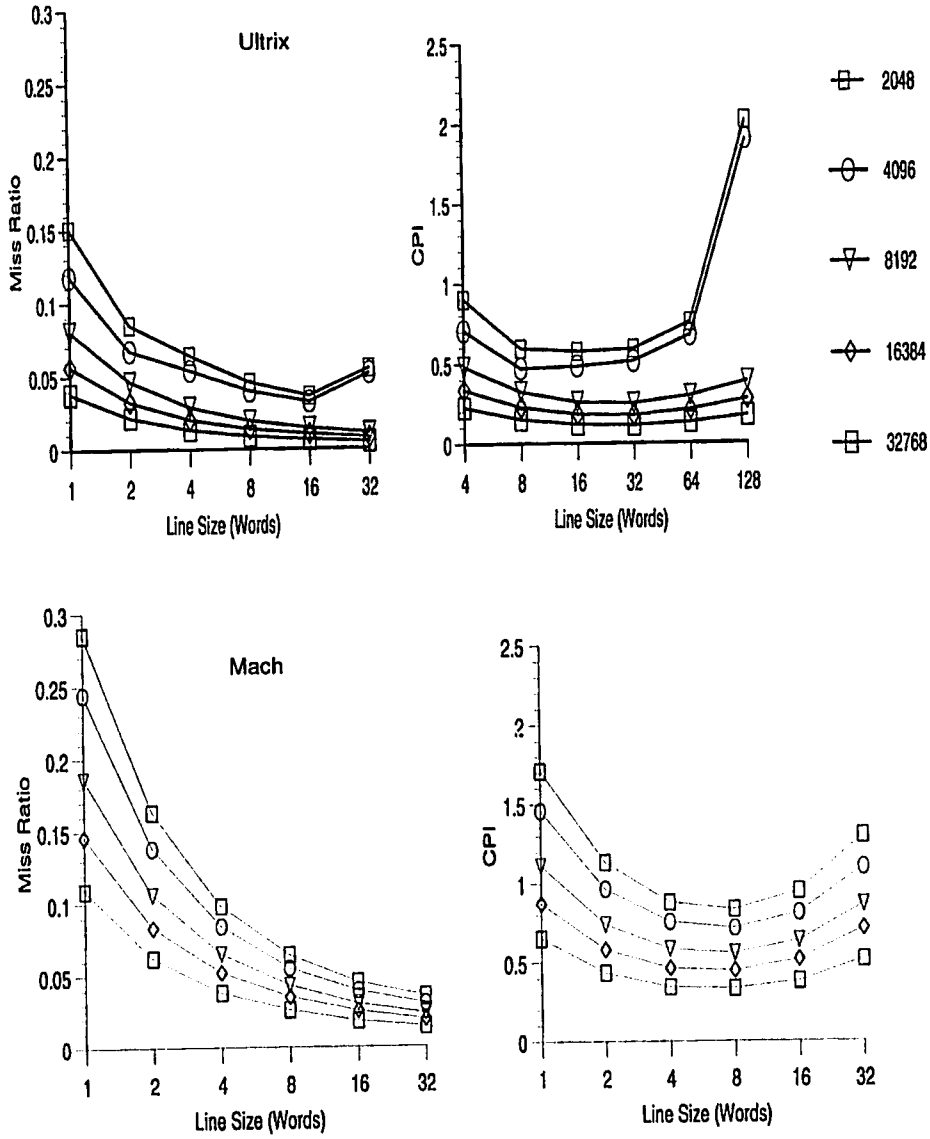


Figure 4.3 Instruction cache performance

This figure plots average I-cache miss ratios and I-cache contribution to CPI for the entire benchmark suite for various direct-mapped I-cache organizations. Notice that cache pollution due to large line sizes is more of an issue for Ultrix than for Mach.

The benchmarks include: IOzone, jpeg_play, mab, mpeg_play, ousterhout and video_play.

miss ratio of 0.013. Note that cache pollution due to large line sizes is an issue when Ultrix runs with small caches.

The bottom left graph in Figure 4.3 (p. 64) shows that Mach generally exhibits higher I-cache miss ratios than Ultrix. For example, an 8K-byte, 4-word-line I-cache configuration under Mach has a miss ratio of 0.065, which is more than double that of Ultrix. The plot indicates that I-cache performance under Mach improves in larger increments (relative to Ultrix) as line size is increased. Further, cache pollution does not occur even for the largest line sizes of 32 words. This suggests that large I-cache line sizes are an effective means for lowering miss ratios under Mach. In fact, doubling the line size is more effective in reducing the miss ratio than doubling the cache size. To see why this is so, recall that Mach has a long instruction path to its user-level API servers. The instructions along this path typically execute only once per invocation of an OS service and have a high probability of being displaced in a small cache before they are used again. Because of this, large line sizes are effective in reducing the average cost to load these instructions into the cache for each use. However, as shown in the CPI plots of Figure 4.3 (p. 64), the degree to which large line sizes can be used without increasing overall CPI is limited due to the additional cycles required to load a large line. For the miss penalties that we selected, I-cache line sizes of 16 words mark an upturn in the CPI plot.

Mach also experiences greater benefits from increased I-cache associativity than does Ultrix. The left side of Figure 4.4 (p. 66) shows that Ultrix exhibits the largest reductions in miss ratio for small caches and primarily when moving from a direct-mapped to a 2-way set-associative I-cache. On the other hand, increased associativity yields benefits over a broader range of cache configurations under Mach. For Mach, highly-associative I-caches can reduce the miss ratio, but cannot completely overcome the problems created by Mach's long code paths. An 8-way, 4K-byte I-cache still has a miss ratio of over 0.03.

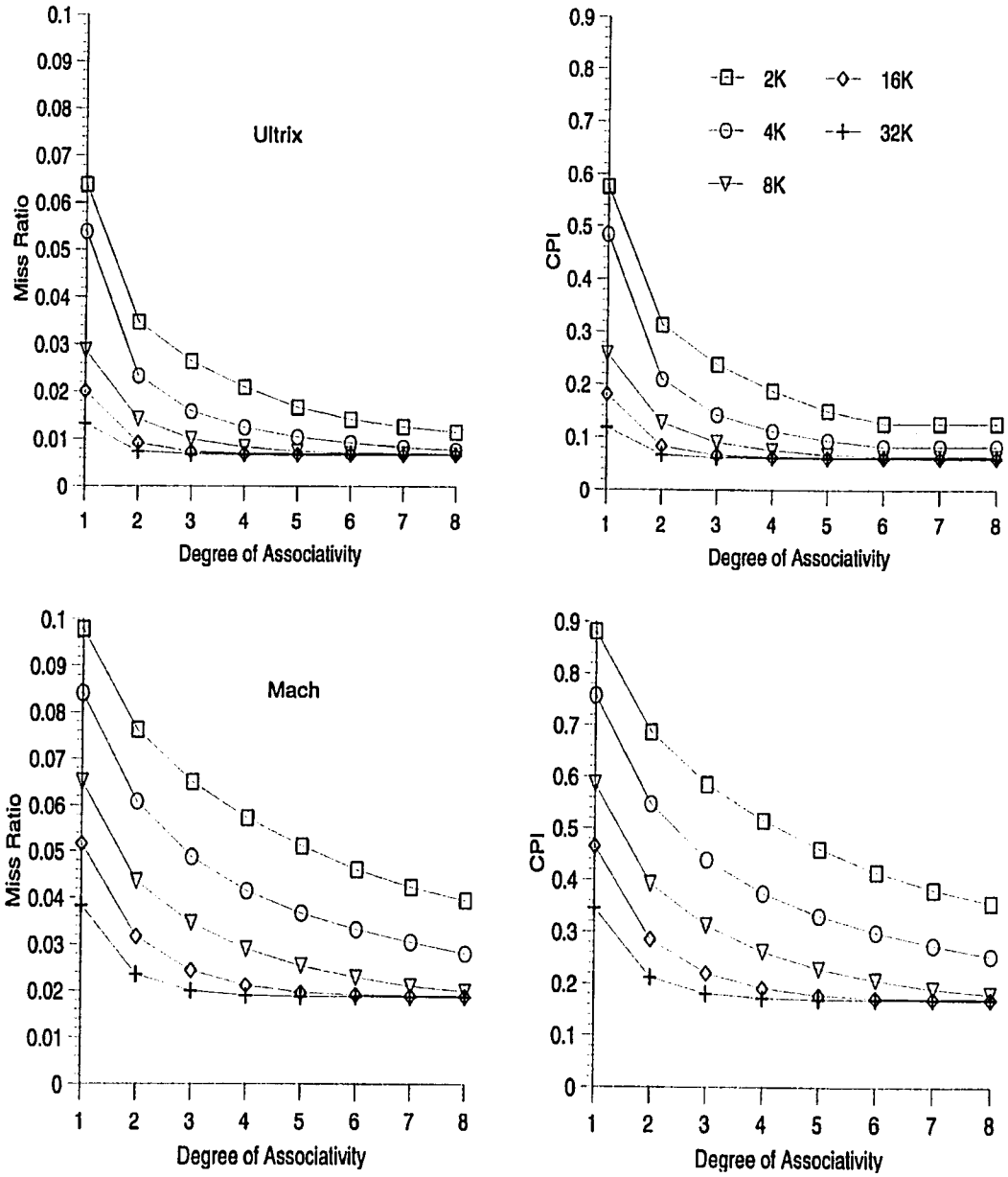


Figure 4.4 Performance of set-associative instruction caches

These plots show increased I-cache performance for various cache sizes and associativities. The line size for both graphs was fixed at 4 words. Increased associativity benefits Mach more than Ultrix over a broader range of cache configurations. The CPI graphs plot the I-cache's contribution to CPI.

The benchmarks include: IOzone, jpeg_play, mab, mpeg_play, ousterhout and video_play.

For small caches, Mach's D-cache miss ratios are also higher than those of Ultrix (not shown). However, unlike the I-cache where longer line sizes and associativity significantly improved miss ratios, D-caches see a more modest improvement. Further, in contrast with I-caches, line sizes greater than 8 words begin to result in D-cache pollution under both operating systems. For our miss penalties, D-cache line sizes above 4 words begin to increase CPI.

4.4 Cost and benefit analysis

The data in Section 4.3 show that tuning the I-cache's line size and increasing associativity can significantly reduce CPI_{instr}. Once system parameters are tuned, however, the most influential component is the cache size. Unfortunately, microprocessor technologies prohibit large on-chip primary caches. Even with sufficient chip area, timing constraints limit the size of on-chip primary caches. Table 4.1 (p. 68) lists a number of current-generation microprocessor cache configurations. The largest primary on-chip cache is the SuperSPARC with 36-KBytes. Even the DEC 21164 with 112-KBytes of on-chip cache can only use 16-KBytes for primary cache.

It is important for architects to evaluate design on-chip memory design trade-offs within a specific chip-area budget. Therefore, we use the I-cache performance results from Section 4.3 and the TLB results from Chapter 3 to evaluate the impact of various memory configurations within a limited area budget. We begin by outlining the area model uses to estimate the cost of various memory components. We then combine the area estimates with performance results to construct a list of the best performing configurations.

4.4.1 Cost analysis

Several cost models have been developed to estimate the die area required for a given memory structure (e.g. register file, cache, TLB, write buffer) [Mulder91, Hill84, Alpert88]. This study uses the MQF model mentioned earlier [Mulder91]. The MQF

Processor	Die Size (mm^2)	I-cache (size, assoc, line)			D-cache (size, assoc, line)			TLB (size, assoc)		
Intel I486DX	81	8-KB	4-way		(unified)			32-U		4-way
Cyrix 486DX	148	8-KB	4-way	4-word	(unified)			32-U		4-way
Intel Pentium	296	8-KB	2-way	8-word	8KB	2-way	8-word	32-I	64-D	4-way
DEC 21064	234	8-KB	1-way	8-word	8-KB	1-way	8-word	32-I	12-D	full
DEC 21164*		8-KB	1-way	8-word	8-KB	1-way	8-word			
Hitachi HARP-1	264	8-KB	1-way	8-word	16-KB	1-way	8-word	128-I	128-D	1-way
PowerPC 601	121	32-KB	8-way	16-word	(unified)			256-U		2-way
MIPS R4000	184	8-KB	1-way	8-word	8-KB	1-way	8-word	96-U		full
MIPS R4200	81	16-KB	1-way	8-word	8-KB	1-way	4-word	64-U		full
MIPS R4400	184	16-KB	1-way	8-word	16-KB	1-way	8-word	96-U		full
MIPS TFP	298	16-KB	1-way	8-word	16-KB	1-way	8-word	384-U		3-way
SuperSPARC	—	20-KB	5-way	16-word	16-KB	4-way	8-word	64-U		full
MicroSPARC	225	4-KB	1-way	8-word	2-KB	1-way	4-word	32-U		full
TeraSPARC	—	4-KB	1-way	8-word	4-KB	1-way	8-word	—	—	—

Table 4.1 On-chip memory in current generation microprocessors

Typical parameters for current on-chip memory structures. These data were collected from a variety of processor data books and issues of Microprocessor Report during the past two years [MReport 92, MReport93]. Throughout this paper we report line sizes in 4-byte words

* The DEC 21164 also has a 96-KByte on-chip second level cache.

model considers the memory cell type (dynamic or static), tag and data bits, organization (fully-associative, set-associative or direct-mapped), drivers and comparators to estimate die area using a technology-independent unit, the register-bit equivalent (rbe). This section summarizes the costs of TLBs and caches as predicted by the MFQ model using the default parameters defined by the authors of the model.

Figure 4.5 (p. 69) graphs the area cost of TLBs with varying degrees of size and associativity. For small, set-associative TLBs (< 64 entries, 1- to 8-way set-associative), increasing the degree of associativity increases the relative die area required. A 16-entry, 8-way set-associative TLB requires 3 times the area of a 16-entry, direct-mapped TLB. For larger TLBs (> 64 entries), associativity has a much smaller impact on die area. For

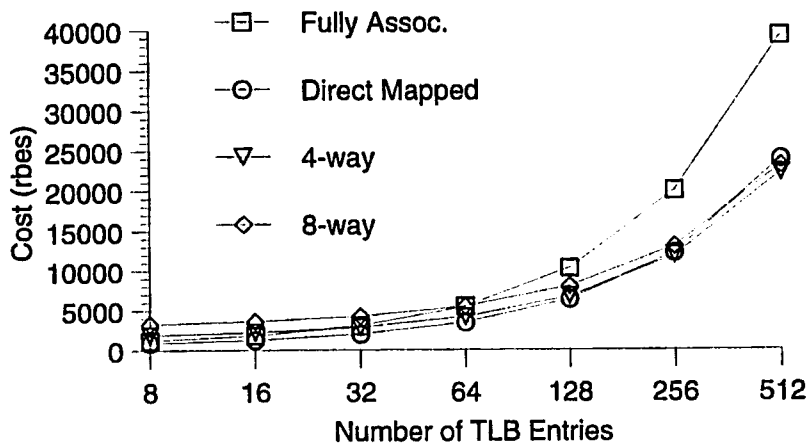


Figure 4.5 Area cost for TLBs of different sizes and associativities

This figure shows the size of various TLB configurations. Fully-associative TLBs require significantly more area than set-associative TLBs. For large TLBs, component sharing (sense amps, drivers, etc.) may actually make some associative structures smaller in area than direct-mapped TLBs.

---

example, with the largest TLB size (512 entries), there is little difference in cost between a direct-mapped TLB and an 8-way, set-associative TLB.

Cost trade-offs also exist between set-associative and fully-associative TLBs (see Figure 4.6). Direct-mapped TLBs are always smaller than fully-associative TLBs. However, for small TLBs (< 64 entries), full associativity costs less than 4- or 8-way set-associativity. For TLBs with 64 or more entries, the opposite is true. In this range, a fully-associative TLB requires twice as much area as a 4- or 8-way, set-associative TLB. For example, for approximately the same cost, designers can choose either a 256-entry, fully-associative TLB or a 512-entry, 8-way TLB.

For larger memory structures, such as caches, there is a different set of trade-offs. Figure 4.7 (p. 71) plots the relationship between area cost and cache organization. Larger line sizes reduce the cost of a cache by as much as 37% when moving from a 1-word line to an 8-word line size. Associativity (not pictured) has a much smaller impact on die area.

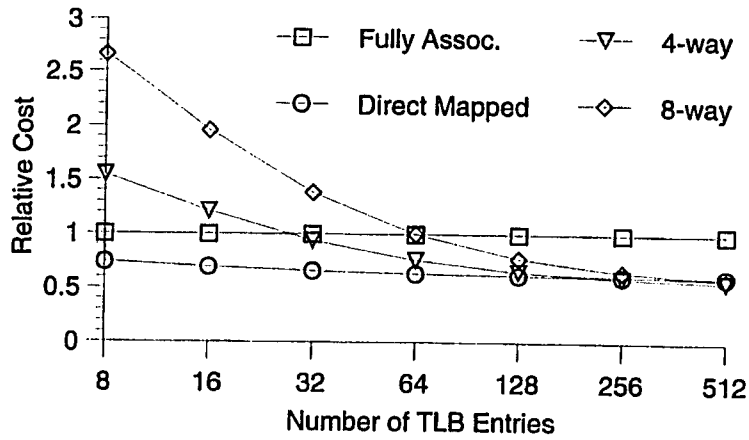


Figure 4.6 Area cost of set-associative TLBs relative to fully-associative TLBs

This graph plots the cost of 1-, 4- and 8-way, set-associative TLBs relative to the cost of fully-associative TLBs of the same size. For small TLBs, it is cheaper to build a fully-associative TLB than to implement 4- or 8-way associative TLBs. For larger TLBs, the trade-offs change; full associativity can cost twice as much as set associativity.

In general, the MQF model gives a good approximation of the total cost for a given memory structure. Mulder at al. compared the model's area predictions against 12 actual processor designs and found typical errors of under 10% and a maximum error of 20.1%. The authors note several limitations of their model. First, it does not consider the relationship between access time and area cost. Second, optimal layout geometry for different memory sizes cannot be completely modeled. Third, changing the aspect ratio can cause the model to underestimate actual area costs. To achieve a higher degree of accuracy, designers should use their own model to determine cost trade-offs within their specific processor technology. For the purposes of this paper, the MQF model is accurate enough to allow us to estimate first-order cost trade-offs in on-chip memory allocation.

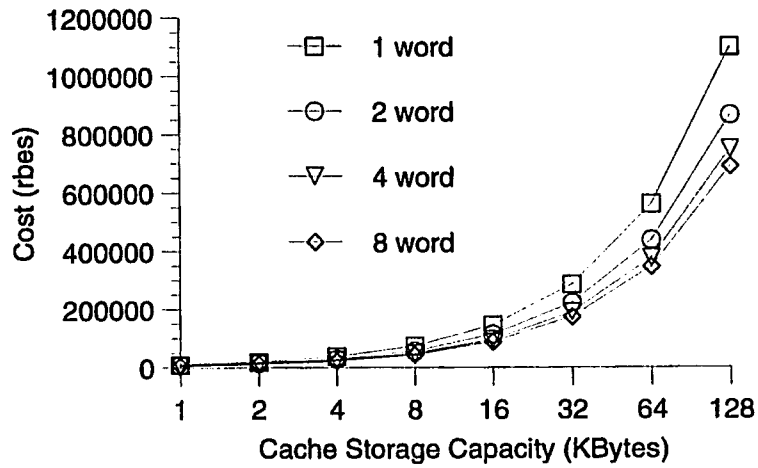


Figure 4.7 Area cost for caches of different capacity and line size

This graph plots the cost of various caches with 1-, 2-, 4- and 8-word lines. Larger line sizes reduce the cost by amortizing the cost of tag and status bits over more data bits.

4.4.2 Performance-driven area allocation

To determine which on-chip memory system configurations have the best performance within the constraints of a fixed, on-chip memory budget, we performed a cost/benefit analysis by combining the cost estimates from the MQF model with our TLB and cache performance data.

We selected a maximum die-area budget by examining various current generation microprocessors (Table 4.1). The data show that most TLBs are between 32 and 96 entries (fully-associative) while on-chip caches do not exceed 32K Bytes (for both instruction and data). The MQF model predicts that the total of these memory structures should cost less than 250,000 rbes. This relatively small amount of on-chip memory reflects technology constraints that will continue for the next several years [MReport93]. High-end systems will provide more on-chip memory, but access times will probably require that this be in a

second-level cache. Moreover, trends towards inexpensive, scaled-back processors such as the MIPS R4200 and the DEC Alpha will keep many processors' on-chip primary caches small.

To illustrate our optimization process, we need CPI values rather than miss ratios or service times. Therefore, we assumed TLB miss penalties to be the same as with an R2000 processor. Because this is a software-managed TLB, miss penalties range from about 20 cycles for misses on user pages to over 400 cycles for kernel-space misses. As noted previously, cache miss penalties were estimated to be 6 cycles for the first word in a line and 1 cycle for each additional word. Of course, different miss penalties will lead to different optimal configurations.

With 250,000 rbe's as our maximum amount of die area for on-chip memory structures, we used the MQF model to determine which combinations of TLB and cache configurations would fit on-chip. The configurations explored are listed in Table 4.3. Next, we used our TLB measurements and cache miss ratios from Mach to compute the contribution to CPI for each configuration, and then sorted the possible combinations by total CPI. The resulting list was very large.

Table 4.8 lists the 10 configurations with the lowest total CPI under Mach. All of the best configurations include a 512-entry TLB. This is due to two factors. First, large

	Total Storage Capacity	Degree of Associativity	Line Size (words)
TLB	64 entries up to 512 entries	1-, 2-, 4-, 8-way and fully-associative (up to 64 entries)	———
I- and D-cache	2K-bytes to 32K-bytes	1-, 2-, 4-, and 8-way	1, 2, 4, 8, 16, 32

Table 4.2 TLB and cache configurations considered

The size for each possible TLB and cache configuration was computed using the MQF model. Combinations of I-cache, D-cache and TLB configurations that required fewer than 250,000 total rbes represent feasible allocations of on-chip memory that fit within the design budget.

TLB (size, assoc)		I-cache (size, line, assoc)			D-cache (size, line, assoc)			Total Cost (rbes)	Total CPI
512	8-way	16-KB	8-word	8-way	8-KB	8-word	8-way	163,438	1.333
512	4-way	16-KB	8-word	8-way	8-KB	8-word	8-way	162,497	1.334
512	2-way	16-KB	8-word	8-way	8-KB	8-word	8-way	162,579	1.335
512	8-way	32-KB	16-word	8-way	8-KB	8-word	8-way	249,089	1.335
512	4-way	32-KB	16-word	8-way	8-KB	8-word	8-way	248,148	1.336
512	8-way	32-KB	8-word	4-way	8-KB	8-word	8-way	243,502	1.336
512	2-way	32-KB	16-word	8-way	8-KB	8-word	8-way	248,230	1.337
512	4-way	32-KB	8-word	4-way	8-KB	8-word	8-way	242,561	1.337
512	2-way	32-KB	8-word	4-way	8-KB	8-word	8-way	242,643	1.338
512	8-way	16-KB	16-word	8-way	8-KB	8-word	8-way	167,815	1.339

Table 4.3 The ten best area allocations

The ten best allocations of die area given a budget of 250,000 rbes. Note that the CPI of these configurations only differ in the third decimal place, making them essentially equivalent in terms of experimental uncertainty.

TLBs substantially reduce the TLB's contribution to CPI. Second, large TLBs do not cost very much relative to on-chip caches. For example, a 512-entry, 8-way set-associative TLB costs just 19,000 rbes while an 8K-byte, direct-mapped, 4-word-line cache costs over 74,000 rbes. Providing a large, set-associative TLB improves performance without significantly adding to total die size.

With respect to caches, the top allocation increases the I-cache size at the expense of the D-cache. Further, this allocation actually requires only 163,438 rbe's, 35% less than the maximum number of rbe's and has only 16-Kbytes of I-cache. Costlier configurations, like row four, supply more I-cache capacity, but their line size/associativity trade-offs lower the system's overall performance.

Most of the best performing configurations include a significant amount of cache associativity. However, access-time requirements may prohibit 4- or 8-way set-associative caches. Therefore, we computed another table by restricting cache configurations to set-associativities of 1 or 2. Table 4.3 shows that this restriction increases the best possible

Rank	TLB (size, assoc)		I-cache (size, line, assoc)			D-cache (size, line, assoc)			Total Cost (rbes)	Total CPI
1	512	8-way	32-KB	8-word	2-way	8-KB	4-word	2-way	239,259	1.428
5	512	8-way	32-KB	4-word	2-way	8-KB	8-word	2-way	248,628	1.447
13	512	8-way	32-KB	16-word	2-way	8-KB	8-word	2-way	232,040	1.462
21	512	8-way	32-KB	16-word	2-way	8-KB	2-word	2-way	241,256	1.473
24	512	8-way	32-KB	4-word	2-way	4-KB	4-word	2-way	228,214	1.475
27	256	8-way	32-KB	4-word	2-way	8-KB	2-word	2-way	249,684	1.477
59	64	Full	32-KB	8-word	2-way	8-KB	4-word	2-way	225,438	1.497
61	128	8-way	32-KB	8-word	2-way	8-KB	4-word	2-way	226,971	1.498
73	512	8-way	32-KB	16-word	2-way	8-KB	16-word	2-way	232,117	1.503
77	512	8-way	16-KB	8-word	2-way	16-KB	2-word	2-way	212,442	1.504
92	512	8-way	16-KB	4-word	2-way	16-KB	2-word	2-way	219,138	1.511
99	512	8-way	16-KB	8-word	2-way	8-KB	8-word	2-way	151,875	1.512
113	64	Full	32-KB	4-word	2-way	8-KB	8-word	2-way	234,807	1.516
1529	64	4-way	8-KB	1-word	1-way	16-KB	2-word	1-way	176,909	2.529

Figure 4.8 Configurations that cost under 250,000 rbes

Memory configurations restricted to caches that are 1- or 2-way set associative. This list represents some of the first 113 best configurations and one of the poorer performing configurations (#1529). Configurations with similar features were removed from the list.

CPI to 1.428. Again, in these configurations, TLBs are large and I-caches are typically 2 to 4 times bigger than D-caches.

4.5 Other techniques to improve I-cache performance

One of the limitations of the results from Table 4.2 (p. 72) and Table 4.3 (p. 73) is that I-cache performance is enhanced at the expense of D-cache size. However, a general purpose workstation must support a wide range of applications, including scientific codes that stress the D-cache. While trading D-cache for I-cache will help improve performance for applications which rely heavily on the OS, it will significantly impact the performance of other applications.

There are other techniques for improving I-cache performance. In this section we examine 4 common techniques: increasing bandwidth, instruction prefetching, instruction bypassing, and pipelining the memory system. Because these experiments only test the instruction stream, we have removed the IOzone, mab and ousterhout workloads and have added several more instruction-intensive workloads including gcc, gs, sdet, nroff, groff. Also, because of technology issues, we assume a two-level cache hierarchy. Level 1 (L1) is the primary cache and Level 2 (L2) is a second level cache.

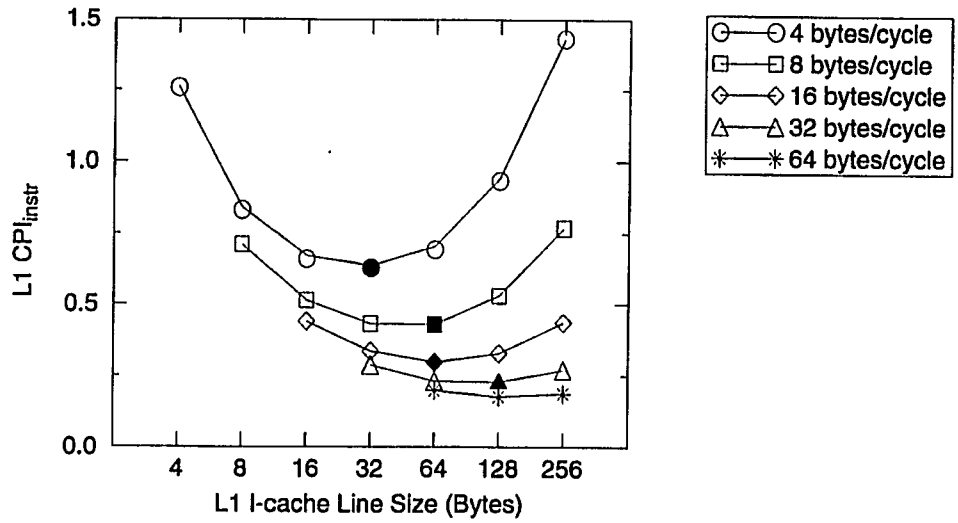


Figure 4.9 L1 CPI_{Instr} vs. line size

L1 contribution to CPI_{Instr} for an 8-KB direct-mapped I-cache backed by an L2 cache with a 6-cycle latency. The data show that as the bandwidth increases, so does the optimal line size (denoted by black symbols). Note that for bandwidths greater than 16 bytes/cycle, the optimal L1 line size is actually greater than the L2's optimal line size. This is due to the large difference in bandwidth and latency between the upper and lower levels of the memory hierarchy.

The execution model for this figure assumes the processor must wait for the entire cache line to refill before it resumes execution.

4.5.1 Bandwidth

Figure 4.6 (p. 70) plots the impact that increasing bandwidth has on L1 cache performance. The plot also shows that a side effect of increasing bandwidth is an increase in the optimal L1 line size (denoted by the black symbols). This benefits cache design in two ways. First, increasing the line size decreases the size of the cache tags. Second, the reduction in area reduces the cache access time. The Mulder area model predicts a 10% reduction in area when moving from a 16-byte to a 64-byte line (8-KB, direct-mapped cache) [Mulder91] while the Wilton and Jouppi timing model shows a 6% decrease in access time [Wilton94].

Number of Lines Prefetched	Line Size (Bytes)		
	16	32	64
0	0.439	0.335	0.297
1	0.305	0.271	—
2	0.270	—	—
3	0.260	—	—

Table 4.4 Prefetching

L1 CPI_{instr} for various line sizes and prefetch lengths. The L1-L2 bandwidth was 16 bytes/cycle and the processor must stall until both the miss and the prefetches where returned to the cache.

The cells with an "—", denote data points that were either not reasonable, or that showed an increase in CPI_{instr}.

The incremental improvements due to increasing bandwidth begin to diminish for rates greater than 16 bytes/cycle. Further, building large cache buses (> 128 bits) can consume a significant amount of chip area and possibly impact the overall cache size. This suggests that once the L1-L2 interface reaches a bandwidth of 16 or 32 bytes/cycle, other techniques might be better suited to improving the L1 cache performance. To investigate this, we fixed the L1-L2 interface at 16 bytes/cycle and used this configuration to examine the effects of prefetching, bypassing and pipelining.

4.5.2 Prefetching

One simple prefetch strategy is sequential prefetch-on-miss, where a cache miss is serviced by returning both the missing line and the next N sequential lines into the cache. Table 4.4 (p. 77) shows that for small line sizes, prefetching can significantly improve performance. The table also shows a result noted by Smith [Smith82]: prefetching over multiple small lines yields better performance than implementing a cache with longer lines. For example, the 64-byte line has a CPI_{instr} of 0.297 while the 16-byte line with 3 sequentially prefetched lines has a lower CPI_{instr} of 0.260. Both configurations return 64

bytes of instructions, however, the longer line size forces the system to fetch more potentially useless instructions. This is particularly true for a miss on the second half of a long cache line because the system must bring in the first half of the line. We have found that when the miss occurs near the end of a line, instructions in the first part of the line are often evicted from the cache before they are referenced. The finer granularity of a 16-byte line overcomes this problem by beginning the fetch much closer to the missing word while allowing the system to prefetch instructions that have a greater potential for being referenced[1].

4.5.3 Bypassing

Sequential prefetch-on-miss can be enhanced by placing the missing line into both the cache and into special bypass buffers. These dual-ported buffers allow the processor to continue execution as soon as the missing word has returned from the L2 cache. One limitation is that while the cache refills, the processor may only fetch instructions from the bypass buffers. Table 4.5 shows CPI_{instr} with and without bypass.

Another policy is to only cache prefetched lines if they are used by the processor. This eliminates any cache pollution due to prefetching. However, our simulations show that this policy does not improve performance over prefetching into the cache. Unless prefetched instructions are used almost immediately, they are likely to be replaced due to the limited number of bypass buffers. Placing the prefetched data into both the cache and bypass buffers increases the chance that a prefetched instruction will be available if it is accessed later in a program run. This is particularly useful for short subroutine calls and forward branches in loops.

---

1. Our results also show that a 64-byte line with 16-byte sub-block placement can perform almost as well as a 16-byte line with 3 line prefetch. On a cache miss, the system only refills the missing sub-block and all subsequent sub-blocks in the line. While the sub-block configuration had more cache pollution, the decrease in refill cost provided the performance gains.

Number of Lines Prefetched	Line Size (Bytes) No Bypass Buffers			Line Size (Bytes) With Bypass Buffers		
	16	32	64	16	32	64
0	0.439	0.335	0.297	—	0.296	0.226
1	0.305	0.271	—	—	0.218	0.224
2	0.270	—	—	0.205	—	—
3	0.260	—	—	0.181	—	—

Table 4.5 Bypassing

This table compares the performance of configurations with and without bypass buffers. The bypass buffers significantly reduce CPI_{instr} by allowing the processor to continue execution as soon as the missing word returns.

For each system, there were as many bypass buffers as lines returned from the memory system (fetched + prefetched lines). The cells with an "—", denote data points that were either not reasonable or that showed an increase in CPI_{instr}.

4.5.4 Pipelining

The final enhancement we investigated is pipelining the L1-L2 interface. This allows the L2 cache to accept a request on every cycle and return a request on every cycle with some latency between requests and refills. During cycles where the processor hits in the cache, the pipeline is kept busy with sequential prefetch requests[1]. Prefetches are stored in a special buffer, called a Stream Buffer [Jouppi90]. The stream buffer is a fully-associative memory with 1 or more lines and is very similar to a bypass buffer.

The results in Table 4.6 (p. 80) show that stream buffers effectively improve I-fetch performance until the stream buffer reaches 6 lines, after which the improvements are marginal. However, stream-buffer performance might be further improved by implementing multiple stream buffers and switching between the stream buffers on

1. Pipelining the memory system also allows data references to be mixed with prefetch requests.

Number of Lines in Stream Buffer	16 bytes/cycle CPI_{Instr}	32 bytes/cycle CPI_{Instr}
0	0.439	0.287
1	0.267	0.186
3	0.184	0.137
6	0.147	0.118
12	0.122	0.103
18	0.114	0.099

Table 4.6 Pipelined memory system with a stream buffer

The L1 cache line size is set by the bandwidth between the L1 and L2 caches (16 or 32 bytes/cycle). This allowed the memory system to accept a request on every cycle.

The model assumes that instructions can be moved from the stream buffer to the I-cache without incurring a penalty. Some implementations may incur a 1 cycle penalty during the move if an instruction fetch cannot be serviced by the stream buffers.

---

subroutine jumps. This would be particularly useful for short leaf-node function calls. Another optimization would be to add a target prefetch table [Smith78]. This table would store the addresses of non-sequential pairs of lines. As every fetch or prefetch is issued into the pipelined memory system, the table is checked to see if there exists an entry. If so, the next prefetch would use the address stored in the table and not a sequential address. We are currently evaluating both techniques.

4.6 Conclusions and future work

OS trends are shifting the way that components of existing computer architectures are utilized. In particular, operating systems trends require more TLB and I-cache support. At the same time, IC process limitations and cycle-time goals are forcing on-chip microprocessor memories to fit into tight design constraints. Our cost/benefit analysis found a number of die-area allocations that provide good support for Mach while staying within a given area budget. In general, the best configurations include large, set-

associative TLBs because they eliminate a substantial component of CPI for relatively little cost and I-cache 2 to 4 times larger than the D-cache. Further, large I-cache line sizes were found to be very effective in reducing both CPI and die area without leading to cache pollution under Mach. We also explored other instruction fetching techniques that do not require a significant amount of chip area, but that nevertheless do effectively increase the performance of a primary I-cache. These techniques reduce the need for larger I-caches, allowing designers to allocate chip area for structures such as D-caches, prefetch buffers and branch prediction units.

This work could be extended in two ways. First, we did not consider the impact of size and associativity on memory access times in a rigorous fashion. An accurate access-time model, such as that developed by Wada et al., could be used to add another dimension to this style of cost/benefit analysis [Wada92]. Second, we only considered the I-cache, D-cache and TLB in die area allocations. A more ambitious study could model the die-area cost and performance benefits of other architectural structures, such as write buffers, pre-fetching units, streaming buffers, branch-prediction units and floating-point units to see if, or to what extent they should be allocated space under a given microprocessor die budget.

CHAPTER 5
Building a Framework for OS/architecture Analysis

5.1 Introduction

Most OS and architecture performance research has focused on isolated areas of the system, illuminating the behavior of some system components, but unable to provide an understanding of how the various components fit into a complete picture of the system. Our work has shown the importance of developing this complete picture. By examining both the OS and architectural point-of-view, we have developed an understanding of how various design issues impact each other and the performance of the system.

This chapter continues the discussion by surveying previous and concurrent work. Most previous work has focused on small portions of the overall system, making it difficult to generalize the results. To help overcome this problem, we have used our work to build an OS/architecture framework which partitions OS/architecture research into three levels. The framework is shown in Figure 5.1 (p. 83).

At the lowest level of the framework hierarchy are hardware structures such as pipelines, caches, and TLBs. These structures must execute the software as efficiently as possible. Above the hardware is the software/hardware interface through which the operating system supports and manages the underlying hardware structures. This is also the level where architectural policies, such as virtual vs. physical caches, constrain the software. At the top of the framework are high-level OS issues that determine the lower levels use and behavior.

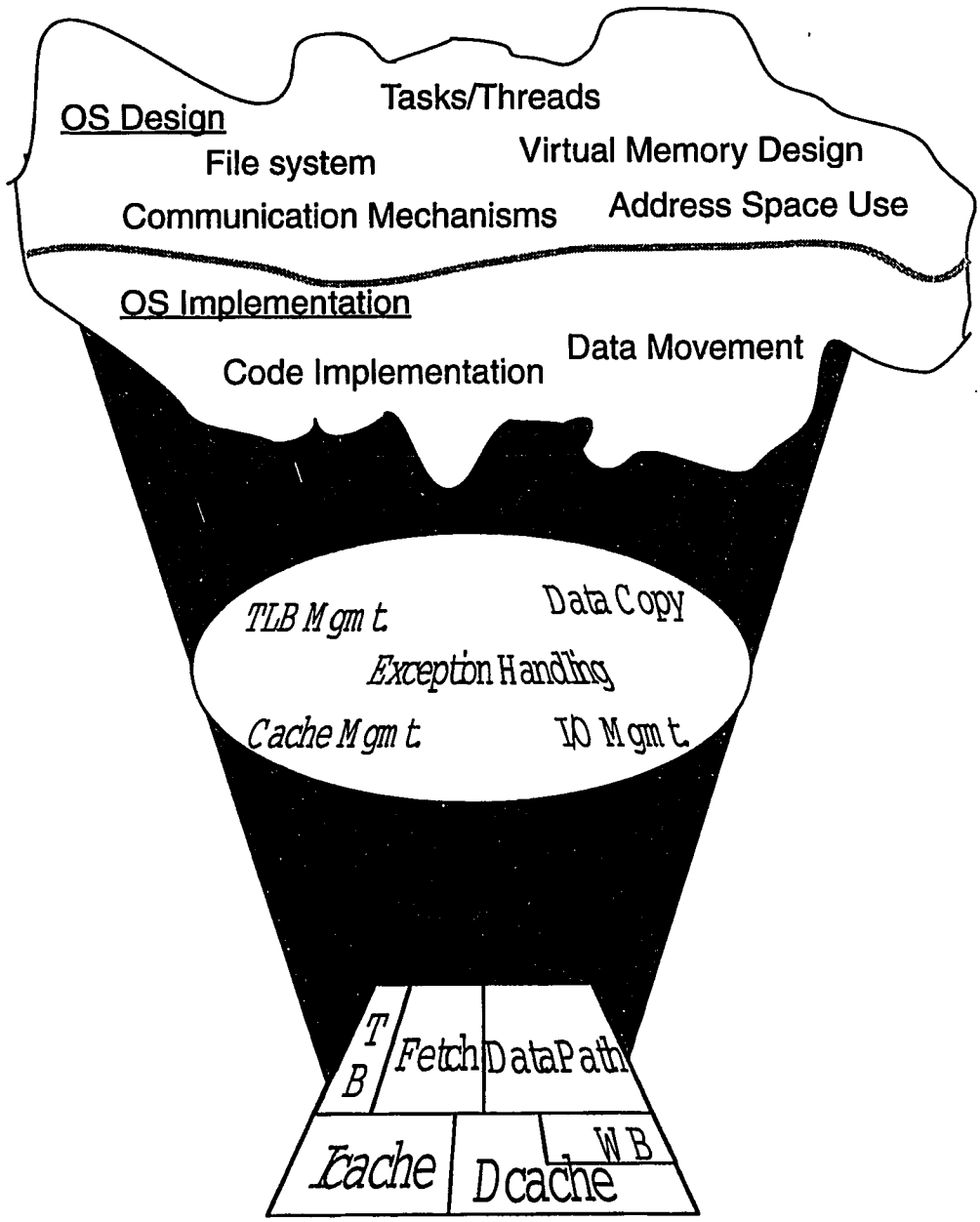


Figure 5.1 OS/architecture framework

There are three levels to the OS/architecture framework. Level 1 contains hardware structures. Level 2 is the hardware/software interface. Level 3 is the higher-level OS design and implementation issues. By examining the behavior and interaction between each level, it is possible to understand the entire system's performance.

Throughout this chapter, we use the framework to structure the discussion, examining where each study fits within the hierarchy, and where works span multiple levels of the hierarchy. We also look for points of commonality and possible trends. The goal is to combine our results with previous and concurrent work into a framework that provides system designers with a better understanding of OS/architecture interactions.

Section 5.2 reviews work at the lowest level of the hierarchy. It is followed by a discussion of the hardware/software interface level and OS implementation issues in Section 5.3. Section 5.4 explores the top level of the hierarchy, tracing various high-level design decisions through the framework to understand how these decisions ultimately effect performance.

5.2 Hardware efficiency

Most architecture studies that consider the operating system have focused on the efficiency of hardware structures [Satya81, Smith82, Emer84, Clark85, Alexander85, Clark85, Alexander86, Clark88, Agarwal89, Flanagan93, Chen94, Cvetanovic94, Maynard94]. Using hardware monitoring or tracing facilities to gather measurements and traces from actual machines, each of these works accurately characterizes the performance of a running workstation. They are also some of the few works that have been able to quantitatively analyze how various architectural design trade-offs impact overall performance.

We begin our discussion by examining various cache performance studies, followed by an analysis of the TLB and other architectural features such as write buffers and multi-issue machines. The results are compared to find points of commonality and possible trends. We also relate the previous results to our own and to three concurrent works [Flanagan93, Chen94, Maynard94].

5.2.1 Caches

Probably the most common result found in many previous studies is that operating system references hurt cache performance. Clark and Agarwal both cite Milandre et al. [Milandre75] and Harding et al. [Harding80] as two of the earliest published works describing this result — a result which has been confirmed by subsequent measurements [Smith82, Alexander86, Agarwal89, Flanagan93, Chen94, Nagle94, Maynard94]. Agarwal notes that both OS references and multiprogramming effects contribute to the difference between user-only and user+OS measurements, with user+OS miss rates typically double user-only miss rates.

Two previous works provide data suggesting that conflict misses between user and OS references are not a significant problem [Agarwal89, Chen94]. In both works, conflict misses between OS and user code usually accounts for between 10% and 20% of the total miss count. Measurements by Flanagan, however, are quite different [Flanagan93]. For Mach 3.0 running the sdet benchmark, 30% to 40% of cache misses where due to conflicts between user and system[1] code, with the percentage increasing with larger degrees of cache associativity.

To understand the different results, we measured the number of OS/user conflict misses across our workload suite. Figure 5.2 (p. 86) shows that the percentage of conflict misses is highly dependent upon the workload. Chen uses benchmarks very similar to groff, real_gcc, and verilog, while Flanagan concentrates on sdet. Therefore, our measurements confirm both Flanagan and Chen's results, showing that they are not truly contradictory.

Our results also show that conflict misses are highly dependent upon the size of the cache (Figure 5.3, p. 87). We believe this is because larger caches reduce the number of

1. Both Flanagan and Chen define system references to include the user-level BSD server and OS kernel.

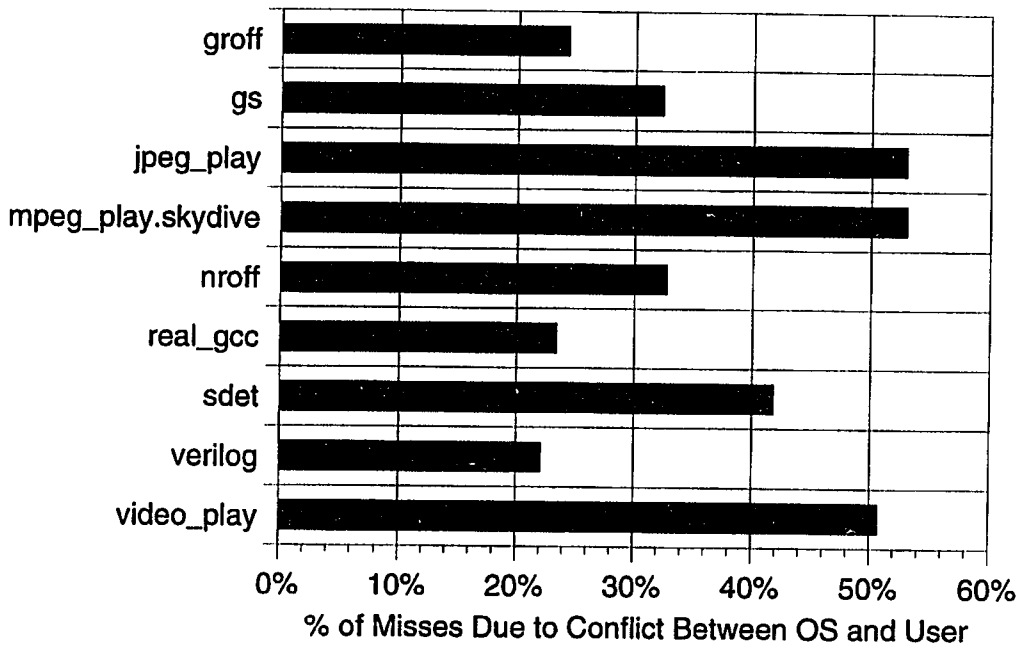


Figure 5.2 Variation in conflict misses

There is a wide variation in the number of misses between OS (kernel + BSD server) and user code. This data was collected from a DECstation 3100 with 64-KBytes of split instruction and data cache, running under Mach 3.0

inter-task conflict misses faster than they reduce the number of intra-task conflict misses. For example, a small 4-KByte cache cannot hold all of the instructions required to perform an RPC under Mach 3.0. Therefore, Mach conflicts with itself as it performs an RPC. A 256-KByte cache can hold the entire path, increasing the intra-task conflicts as a percentage of total misses.

Agarwal's data also showed that OS misses under Ultrix contribute a larger fraction of total misses than under VMS. This is an interesting result because it points to a potential trend: a significant increase in the operating system's contribution to the overall cache miss rate. Just like Ultrix and Mach, VMS and Ultrix are fundamentally different operating systems. VMS places all system services and system commands inside the operating system kernel. Ultrix, a BSD version of UNIX, places many system commands

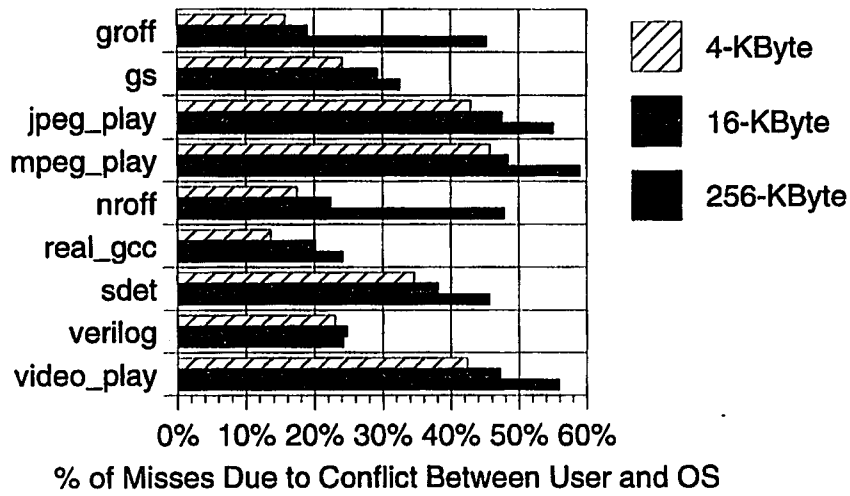


Figure 5.3 Variation in conflict misses across cache sizes

The number of OS/user conflict misses increases with increasing cache size.

in applications programs like cat, grep, and ls. This difference represents a *decomposition* of OS functionality, shifting services outside of the kernel and into user space. Our results show a similar trends between Ultrix and Mach 3.0 (Figure 5.4, p. 88).

Many of the works also evaluate how various cache parameters (size, associativity, and line size) impact OS and user codes. Agarwal shows that even with large caches (1-MByte), both user and system references (instruction and data) have a miss rate of at least 1% [Agarwal]. However, this result may be due to the limited size of the traces[1] (< 512-K references). [Flanagan93] uses much longer traces (> 1 billion references) and shows that a 256-KByte, 8-way set-associative cache can reduce the combined I- and D-cache miss rate to 0.33%. [Maynard94] shows similar results for instruction caches. However, Maynard et al.'s D-cache results measure a 1-MByte D-cache miss rate over 2% for commercial workloads such as TPC or SDET.

1. Agarwal uses a technique called "trace stitching" to create a long trace from shorter segments.

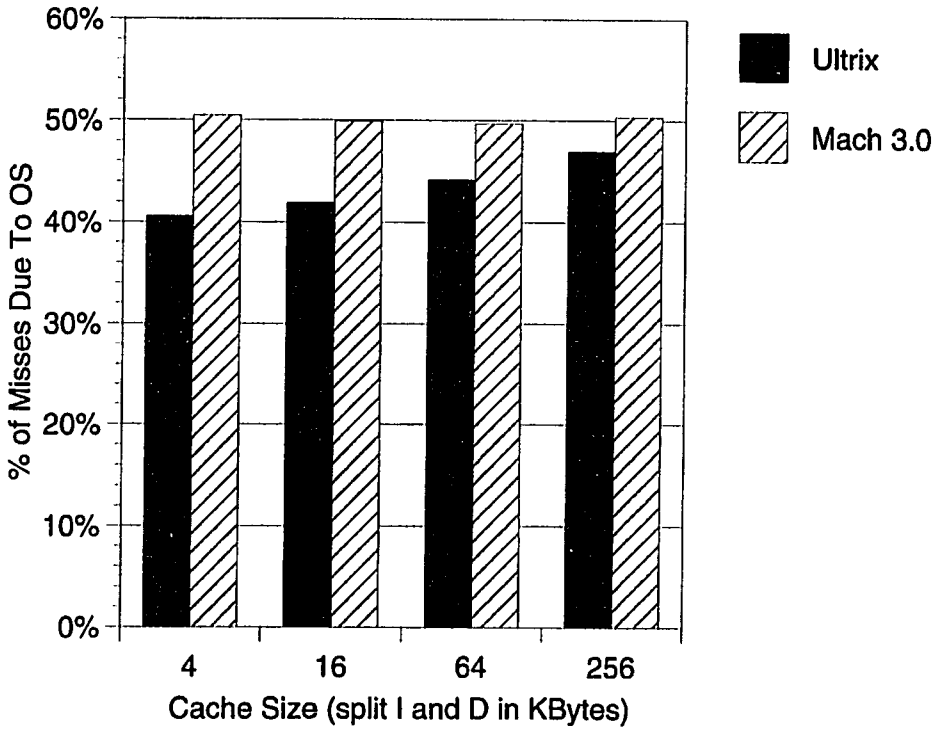


Figure 5.4 Percent of misses due to OS

For each cache size, the percentage of total misses due to Mach 3.0 (kernel + BSD Server) is larger than for Ultrix. However, as cache size increases, the difference between Ultrix and Mach decreases.

For smaller caches, in the range of 8-KByte to 32-KByte, Agarwal and Flanagan both show direct-mapped unified cache miss rates between 2% and 7%. Maynard et al.'s commercial workload results are much more pessimistic, with I-cache miss rates between 4% and 22%, and D-cache miss rates between 4% and 10%. Our own results show miss rates similar to Agarwal and Flanagan. The difference between Maynard et al.'s data and our own results can be traced to differences between the workloads. Commercial benchmarks such as TPC, LADDIS require significantly more memory than the workloads we have used[1].

---

1. This is not to suggest that our workloads do not stress the memory system. It only shows that there are even bigger workloads.

Comparing the work of Smith and Clark with our work and that of Chen and Maynard et al., we see a shift in the relative importance of I- and D-caches [Smith82, Clark84, Chen94, Maynard94]. Smith and Clark both see more D-cache than I-cache stalls while the other works show that I-cache stalls are significantly greater than D-cache stalls. Our own data in Chapters 2 and 4 show that the I-cache is responsible for the largest portion of stall cycles. In Maynard et al.s' data, the I-cache miss rate is typically twice the D-cache miss rate for caches in the 8-KByte to 32-KByte range. Because there are many more instruction references than data references, the I-cache miss rate will be the dominant factor for small caches. This is an important result because most processors provide a very small on-chip level 1 instruction cache (see Table 4.1, p. 68).

There are two reasons for this shift: 1) the OS and software trends discussed in Chapter 4; 2) Smith and Clark's results are from a CISC architecture. Since CISC machines execute far fewer instructions that RISC machines [Bhandarkar91], it is possible that architectural differences influence the relative importance of the I-and D-caches.

There is also some agreement on the effects of tuning the cache line size. Using a combined I- and D-cache, Agarwal shows that increasing line size does improve the cache miss rate. However, the largest benefit from longer line sizes is seen in larger caches (> 32-KBytes), a result influenced by the unified cache used in [Agarwal89]. The measurements of Maynard et al. and our own measurements distinguish between instruction and data references. For I-caches both works show a significant improvement in overall miss rate with increasing line size[1]. For the D-cache, Maynard et al. shows only a small improvement in miss rate for line sizes greater than 16 Bytes.

---

1. Our results for Mach are similar to Maynard et al., for Ultrix, small I-caches cannot tolerate large line sizes (> 8 Bytes).

5.2.2 TLBs

In general, TLB stall cycles account for a small portion of a program's total execution time (usually under 10%) and have fairly low miss rates (usually under 4%). Clark's studies show TLB misses consume 6% to 8% for the VAX-11/780 and 4.6% for the 8800 [Clark85a, Clark88]. The reduction is due to the increased size of the 8800 TLB— the VAX-11/780 has an 128-entry TLB, the 8800 has a 1K-entry TLB. Clark et al. note that while the 8800 TLB is 8 times larger than the VAX-11/780, there is only a 1/3 reduction in TLB stall cycles. They hypothesize that the small decrease may be due to software's increasing code and data size. Our measurements confirm this hypothesis, showing that, on average, TLB stall cycles account for 2% of our workloads'[1] run times under Ultrix and 8% under Mach.

[Satya81, Alexander85, Clark85, Chen92, Talluri92] show that TLB size is an important consideration. Until a workload's working set can be mapped by the TLB, TLB stall cycles will significantly impact overall performance. Associativity can eliminate a significant number of conflict misses. However, because of the coarse granularity of a PTE entry[2], TLBs are much more sensitive to small amounts of associativity than caches. For example, Clark and Emer found that a 2-way set-associative TLB performs as well as a direct-mapped TLB that is 4 times larger [Clark85].

Several studies reveal a large variation in TLB service cost. The VAX-11/780 average TLB-miss service time is 21 cycles [Clark85]. Huck and Hays and our work show the average ranging from 10 to 60 cycles depending on various implementation issues [Huck93]. Some works estimate that the increasing disparity between cache and main memory access times will increase the cost of a TLB miss to between 60 and 100 cycles

---

1. Averaged over the workloads, groff, gs, jpeg_play, kenbus, mpeg_play, nroff, ousterhout, real_gcc, verilog and video_play.

2. On today's' processors, each TLB entry typically maps at least 4-KBytes of memory.

[Chen92, Talluri94]. Without careful consideration of TLB management design, it is possible that future systems could see these large miss penalties. However, better TLB designs aimed at reducing the TLB miss rate could alleviate some of the performance impact.

There are some applications where TLB performance dominates the overall performance. Talluri's simulations show that coral, nasa7 and compress can spend 50%, 40% and 26% (respectively) of total execution time servicing TLB misses [Talluri94]. Graphics applications can also see significant TLB performance problems. McCormack describes how drawing a line in the X display server can touch hundreds of 4-KByte pages, each of which must be mapped by the TLB [McCormack91]. Because little time is spent on any one page, TLB misses become a significant portion of the total cost. McCormack suggests that the TLB provide a special TLB entry "dedicated to mapping a frame buffer into virtual memory, " in effect, providing architectural support for multiple page sizes. The idea of architectural support for multiple page sizes have become very popular in recent microprocessors and we will continue discussion of this in Section 5.3 (p. 93).

In general, all of the studies have shown that TLBs are a small but important aspect of architectural performance. Subtle changes in the architecture or software structure can significantly increase the cost of TLB management. While most measurements show that TLB performance usually accounts for 10% or less of total execution time, TLB misses account for up to 20% of the machine's stall cycles. Further, architectural advances such as multi-issue are increasing the relative importance of the TLB by increasing the number of instructions executed per cycle. It is important that TLB designs and their impact on OS technologies be continually re-evaluated to insure that TLBs consistently deliver good performance.

5.2.3 Other hardware issues

There are several other architectural components that can influence performance. Chen's results demonstrate that OS references stress the write buffers more than user applications [Chen94]. This is understandable given the role of the OS as a mover of data. Graphics services also heavily utilize the write buffer. In the R4200 processor, a low cost version of the MIPS R4000, the write buffer design was modified to meet the graphics needs of the WindowsNT operating system. It also contains an I-cache twice as large as the D-cache, again to support Windows NT (see Table 4.1, p. 68).

Other potentially important architecture trends include superpipelining, multi-issue (superscalar and VLIW), multithreaded and multiprocessor architectures. Many works have examined OS interactions with multiprocessors, but most have focused on numerically intensive workloads. Few have examined how the other architectural trends influence OS design and performance. It is possible that superpipelining and multi-issue might decrease OS performance. Or, superpipelining and multi-issue could improve OS performance, especially if the compiler can better schedule OS code. The effects of newer hardware structures, such as branch prediction, is also unknown. As an example, the effectiveness of branch prediction methods that store extra state in the instruction cache might need to be re-evaluated if OS I-cache performance continues to decline.

5.2.4 Summary of hardware issues

It is clear that caches are the most influential hardware structure for OS and overall performance. The increased modularity, functionality and portability of software affect performance by reducing the effectiveness of caches. Operating systems that neglect to consider how their design and implementation impacts the cache could encounter significant performance problems.

TLB performance can also be a problem. Software's growing trend towards sparse address spaces significantly reduces the ability of a TLB to completely cache a software system's working set. However, multiple page sizes and Talluri's subblock TLB work show that hardware can support software trends if we understand how the trends utilize the hardware [Talluri94].

OS designers might spend more time tuning OS code to leverage architectural advancements. Once commonly used code paths are located, OS designers can tune the code paths to the architecture. One should ensure that frequently used code paths do not map to the same set of cache lines, causing a performance drop because of conflict misses. For example, Tapeworm II's performance improved 33% by rearranging code to reduce cache conflicts in critical sections of the simulator [McFarling89, Uhlig94c].

5.3 Direct hardware/software interactions

The hardware/software interface forms the middle portion of our framework. Its main purposes are to define how the hardware and software communicate, the amount of OS support necessary to manage the hardware, low-level virtual memory and protection mechanisms and the ability of the OS to efficiently provide higher-level functionality. It is also the place where OS and hardware designers can provide direct support for each other.

The first part of this section focuses on hardware management and policy issues where the OS directly interfaces to the architecture. This is followed by a discussion of OS techniques that avoid costly hardware functions and OS techniques which can improve the hardware's performance.

5.3.1 Hardware management and policy

5.3.1.1 TLB design and management

There are a wide range of TLB and virtual memory designs. Some architectures provide software-managed TLBs, others have hardware-walkers, while a few use both. Some architectures provide fine-grained control over page size while others only provide small and large page sizes. This diversity is a problem for operating system designers because each system requires its own special memory management code. It also underscores the fact that there does not seem to be a single "good" approach to virtual memory support.

One of the most basic differences is hardware vs. software TLB management. The argument against software managed TLBs is that hardware can more efficiently service a TLB miss. However, the results in Chapter 3 show that if the architecture provides sufficient support, software-managed TLBs can have very good performance. Further, as memory latencies increase, the most significant component of a TLB miss handler could become the cost of fetching a page table entry from memory. Software-managed TLBs also provide a greater degree of freedom in determining page table structure and TLB caching policy. This allows the OS to optimize the page table structure and TLB replacement policy for a given software environment.

Another issue is the TLB's replacement policy. Traditionally, we think of replacement policies such as first-in first-out, least recently used, or random. For TLBs, Uhlig et al. have shown that replacement policy does not significantly impact TLB performance [Uhlig94b]. It is important, however, that the architecture allow the OS to manage the replacement of performance-critical PTEs. In architectures like the MIPS, this is accomplished by partitioning the TLB. Chapter 3 showed that the number of special hardware slots can influence performance. Subsequent MIPS processors, like the R4000, allow the operating system to determine the specific partition point. An alternative solution

is to provide a special lock bit on each TLB entry, allowing the OS to dynamically adjust the number of special entries to best suit the needs of the workload.

Many previous TLB designs did not provide space for process identifiers (PIDS). Without PIDS, the TLB must be flushed on a context switch. Clark et al.'s measurements showed that the impact of TLB flushing varies widely: anywhere from 0.4% to over 80% of all TLB misses are due to flushing [Clark85a].

Another issue is how the TLB maps the operating system. Systems such as the VAX-11/780 split the TLB into two sections, user and system [Clark85]. The VAX-11/780 maps the operating system while other architectures, such as MIPS, place the OS in unmapped space, allowing all OS text and pre-allocated data references to bypass the TLB. Other architectures use multiple page size TLBs to map the OS. Both the DEC Alpha and IBM PowerPC provide a special "super-page" TLB slot to map the operating system.

Architectural support for multiple page sizes has become very popular in recent microprocessors. In the simple case, a few TLB entries are dedicated to mapping specific software modules, such as the operating system kernel or X display server. The PowerPC architecture provides a special BAT TLB which holds a few large pages managed by the operating system [IBM94]. If the MIPS R2000 architecture provided this feature, mapping the X and BSD servers could have reduced the growth in expensive TLB misses seen with Mach 3.0.

Multiple page sizes can create page management problems for the operating system. Talluri has shown that the overhead associated with changing the operating system's VM code, the increased cost of page table management, and the time required to effectively utilize multiple page sizes can be prohibitive [Talluri94]. However, Talluri also shows that small architectural changes in the TLB design significantly reduce the operating system burden. This allows the system to use large pages when it is cost effective while not penalizing the system when the cost is prohibitive.

5.3.1.2 Exception handling

There are three important issues to consider in exception[1] handling: 1) the number of exception vectors; 2) the amount of information the architecture provides the operating system and; 3) the amount of hardware state and processing the OS must manage. Each of these impacts the performance of the system by influencing the amount of OS code and data movement.

Chapter 3 showed that the architecture should supply enough exception vectors so that the operating system can provide efficient handlers. This is best done by architecting a unique vector for every type of exception. While some may consider this overkill for current OS and software, it does allow the system to grow, possibly avoiding performance problems in future OS technologies.

It is also important for the architecture to supply enough information to the operating system. Anderson discusses how, during a page fault, the Intel i860 processor provides neither the type of page fault nor the faulting address [Anderson91]. This forces the operating system to spend a considerable amount of time deriving this information (26 additional instructions according to Anderson). The page fault could be serviced more quickly if the hardware simply provided the relevant information.

Architects also need to consider the amount of state the OS must manage during a kernel crossing. Software exposed pipeline registers, such as those in the Motorola 88000, or large register sets like the SPARC register windows, slow down a kernel crossing or context switch [Anderson91]. Some architectures (e.g., MIPS and PowerPC) reserve registers for the operating system, allowing the OS to sometimes avoid the expense of saving and restoring a task's register set.

---

1. The term exception has never been well defined by the architecture community. We refer to an exception as any event that causes the machine to enter the operating system. This includes TLB misses, floating point exceptions, and device interrupts.

Finally, it is important to note that architectural trends such as multi-issue and superpipelining are increasing the number of instructions that a processor executes concurrently. Since most processors will abandon all uncompleted processing at a kernel entry, there is the potential to waste many cycles of computation. While this has not been a performance problem in the past, increasing concurrency may reach the point where this does become a performance issue.

5.3.1.3 Cache management

To guarantee consistency between the caches and main memory, operating systems must sometimes flush part or all of the cache. This is true for both physically and virtually indexed caches. Some flushes are unavoidable, such as flushing a write-back cache before a direct memory access (DMA) write. Other times, intelligent OS design can circumvent or minimize the problem.

There are two basic management issues for physically-indexed caches. When a physical page is initially mapped into the virtual space, the OS must insure that the cache does not contain any references to that page. UNIX implementations that zero fill a data page when allocating it to a task avoid this problem because the zero fill code (bzero) flushes any inconsistent lines from the D-cache. Newly allocated instruction pages however, must still be flushed. DMA I/O can also create inconsistencies. Write-back caches must be flushed before a DMA write to insure that main memory contains an up-to-date copy of the data. Both write-back and write-through caches must be flushed before a DMA read to guarantee that the cache will not contain stale data after the read is complete.

Virtually-indexed caches present a number of problems for the OS. Unlike physical caches, which have a one-to-one mapping between cache and memory, virtual caches can have a many-to-one mapping between cache and memory. Commonly referred to as *aliasing*, this ability for the cache to contain multiple copies of a memory location can create data inconsistencies. Figure 5.5 (p. 98) illustrates this problem.

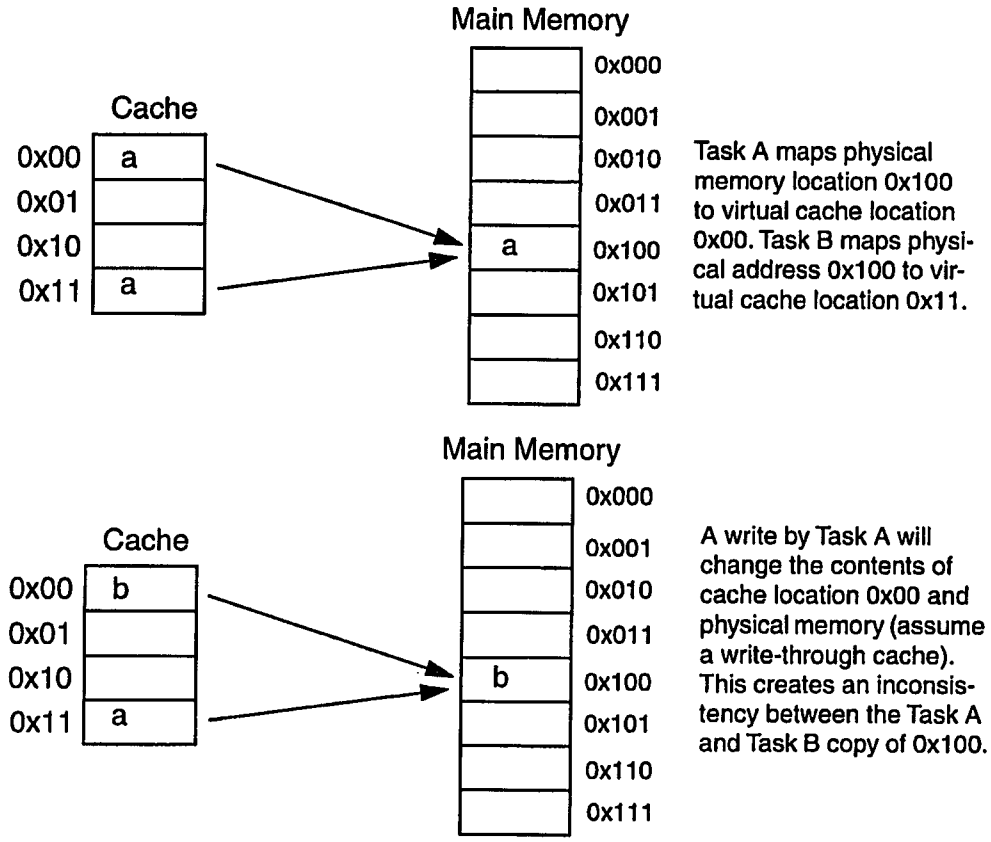


Figure 5.5 Illustration of inconsistency in virtually indexed caches

Aliasing can create performance problems. For example, copy-on-write improves task creation performance by allowing the parent and child tasks to share one copy of text and data (see Section 5.3.2, p. 99). However, unless the two sets of virtual addresses (parent's and child's) map to the same cache lines, the two tasks will each cache separate copies of the text and data. This increases the number of cache misses for two reasons. First, the child task experiences compulsory misses while loading the text and data into the cache. Second, the child task might displace part of the parent's cache state, further increasing the cache miss rate.

To overcome this problem, the operating system's page allocation policy can carefully map virtually-shared memory to the same cache lines. This requires aligning the

pages modulo the cache size [Chao80, Inouye, Wheeler92]. Chao et al. and Wheeler and Bershad present other techniques which allow the operating system to avoid or postpone cache flushing. Both show that these techniques allow virtual caches to provide performance similar to physically addressed caches. However, Chao et al. argues that operating systems running on large, linear addressed spaces with virtual caches should use shared memory to support memory sharing.

5.3.2 Avoiding costly hardware

The performance of some hardware mechanisms, such as memory copy, are difficult to improve. Therefore, operating system designers have avoided copying data by implementing software mechanisms that map the same physical memory into multiple virtual address spaces. This *memory mapping* occurs without the knowledge of an application, preserving a system's semantics while avoiding unnecessary work [Accetta86].

For example, when a UNIX task (process) forks, the semantics call for a complete copy of the task to be created in its own address space. With memory mapping, the operating system can virtually copy each page by marking every page read-only. Now, only pages that are modified will be physically copied. In the worst case, a task will force the operating system to copy every data page. However, the text pages can be shared by both tasks. Memory mapping is also used to pass data between tasks. This allows one task to fill a region of memory with data and then to pass the data to another task without forcing the data to be copied.

Stodolsky et al. describe another software technique that avoids costly hardware operations [Stodolsky93]. Operating systems often use interrupt masking to protect critical sections of the operating system. However, architectures provide minimal interrupt mask support, making it a costly operation. Relying on the fact that interrupts during critical sections are infrequent, Stodolosky et al. set a software interrupt mask to indicate

which interrupts need to be masked. Setting a software interrupt mask is an inexpensive operation. The general case, when a critical section is not interrupted, can proceed at full speed. In the special case, where a critical section is interrupted, the OS must provide a special handling mechanisms. However, this cost remains relatively small relative to the cumulative cost of masking hardware interrupts for every critical section.

5.3.3 OS strategies to improve hardware performance.

The operating system's role as manager of hardware structures gives it an enormous amount of control over the hardware's performance. Some OS functions, such as the operating system's memory routines (e.g., bcopy, bcmp), have very explicit control. Other components, like page-mapping algorithms, have more subtle, implicit effects. This section examines how the operating system's hardware management functions influence overall performance

Memory copy, fill, and compare functions are some of the most carefully tuned components of an operating system. Using code techniques such as loop unrolling and strength reduction, and an intimate knowledge of the ISA, write buffers, and cache sizes, these functions attempt to deliver the maximum possible memory speed.

Operating systems' also utilize RAM to help reduce the cost of I/O operations. When possible, small I/O requests are buffered in special kernel RAM caches. This allows the operating system to more efficiently access the I/O device. Most operating systems also provide a RAM based file system cache to store frequently accessed disk blocks. Multiple requests to the same file can be serviced from the file cache, completely avoiding a costly disk access.

Because of the high cost of accessing a disk, operating systems also employ scheduling algorithms. The disk scheduler combines multiple disk requests, attempting to maximize the amount of data that can be read from or written to a disk. Brusch and

Kondoff have shown that disk scheduling techniques can increase the file system performance several hundred percent [Busch85].

There are other, more subtle, ways in which operating systems can influence hardware performance. Sites and Agarwal observed that physically-indexed caches usually performed worse than virtually-indexed caches in low multiprogramming environments [Sites88]. The loss in performance is due to the conflict misses introduced by poor page mappings. Kessler and Uhlig have also observed this effect [Kessler91, Uhlig95]. Uhlig's results, for several of our workloads, are shown in Figure 5.6 (p. 102). Kessler's work showed that careful page-mapping algorithms can reduce the number of total cache misses by as much as 55%.

Torrellas has also examined conflict misses, focusing on code layout in the operating system kernel [Torrellas95]. This optimization is very important to OS performance because bad conflicts cannot be removed by a page mapping algorithm. Torrellas has shown that carefully arranging OS code to minimize the number of conflict misses reduces the I-cache miss rate by 30% - 80% and improves overall execution time 7% - 15%.

Another technique the operating system can use to improve hardware performance is to not cache references that will not benefit from caching. For example, memory operations that are much larger than the cache cannot benefit from being cached. They will, however, have the effect of flushing the cache. McCormack reports that caching the DECstation 3100's frame buffer[1] usually reduced the overall performance [McCormack91].

1. The DECstation's D-cache is 64-KBytes. The frame buffer is 864-KBytes.

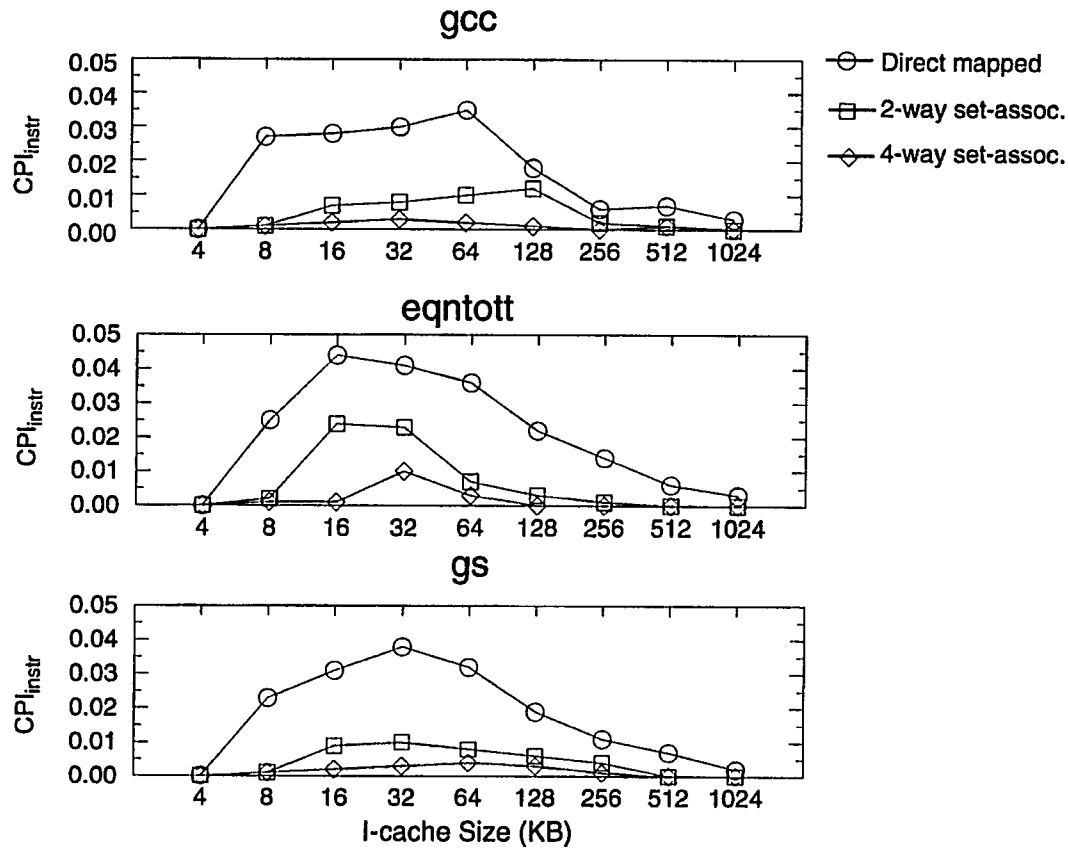


Figure 5.6 Variability in CPI_{Instr} versus I-cache size and associativity

These plots show variability in performance over multiple runs of the same workload in a physically-indexed I-cache. Performance varies because the allocation of virtual pages to physical cache page frames is different from run to run. Variability is reported on the y-axis in terms of one standard deviation of CPI_{Instr}. The plots are reproduced from [Uhlig95].

5.3.4 Summary

The hardware/software interface presents a number of issues for both OS and hardware designers. Some are obvious, such as TLB miss service time, while others, like the effects of conflict misses in the cache, are more subtle and difficult to quantify. Most influence both the architecture and the operating system. Architecture and OS designers

need to understand this interface, and how it fits into the entire framework, in order to design better computing systems.

5.4 System software issues

At the top level of our analysis framework (Figure 5.1) are the various OS implementation and design issues that influence the behavior and performance of the system. Unlike the previous layer, where there was a direct connection between software and hardware interactions, this layer has few direct connections, making it more difficult to reason about how design trade-offs will impact overall performance. The system software issues layer is broken into two distinct components: OS implementation and OS design. While the boundary between implementation and design is sometimes fuzzy, we separate the two by distinguishing between the types of abstractions the OS presents and how those abstractions are implemented.

5.4.1 OS implementation

Software engineers are constantly developing new techniques for implementing software systems. However, at the most basic level of software performance, there are just two main issues: 1) how many instructions are executed and 2) how much data is touched. This section focuses on these issues, using several examples to illustrate how OS performance can be improved.

5.4.1.1 Coding strategies and implementation

Producing efficient code is essential to achieving good OS performance. The size and complexity of the operating system, however, can make this a difficult task. Porting OS code between architectures exasperates this problem because hand coded routines optimized for one system may not perform well on other systems. Further, OS code has a very long life. Often a software designer uses implicit information to optimize a software

module. The implicit optimization can be lost by subsequent code writers who do not understand the rational for the original code's structure.

One example of how code porting can result in poor performance is the Ultrix UDP/IP research by Kay and Pasquale [Kay93]. Measuring the performance of a DECstation 5000, Kay and Pasquale showed that the operating system software prevented the network hardware from achieving its maximum performance. While some performance loss due to software overhead is expected, their measurements showed UDP/IP throughput was 1/5 the expected rate. Using good analysis techniques, Kay and Pasquale discovered that the UDP/IP checksum algorithm was the problem. Carefully written to exploit CISC architectural features, it was ill suited for the RISC-based DECstation. By rewriting the code to take advantage of the RISC Architecture, Kay and Pasquale improved UDP/IP throughput 33%.

Another example of poor code performance can be found in the Andrew File system's (AFS) local cache manager. AFS is a distributed file system that caches recently used files to improve file system performance. Using the local disk to cache files should allow AFS to provide cached-file performance near that of the local file system. Stolarchuk, however, discovered that AFS code increased the cost of an AFS cache access by 300% over the local file system [Stolarchuk92]. Modifying the code to utilize an optimized, common-case code path, brought the performance of cached AFS files to within 10% of the local file system's performance.

Code tuning can eliminate some performance problems. However, overall code structure often limits system performance. In these cases, a more radical restructuring of the code, possibly adopting a different algorithm or paradigm while maintaining the higher-level abstraction(s), can improve performance

For example, tuning the thread management routines can improve Mach 3.0's performance. However, Draves et al. have shown that a more radical change in thread management can also improve performance [Draves91]. Typically, a thread that blocks

inside the kernel must have its register state and stack saved before the OS can switch to another thread. Using continuations to reduce the amount of state that the kernel must save and restore when a thread blocks, Draves et al. were able to reduce the amount of thread state by 85% and improve RPC performance by 14%.

A more radical restructuring technique is the migrating thread model used by Ford and Lepreau to improve the performance of user level services in Mach 4.0 [Ford94]. Unlike many operating systems, where a thread must be "awakened" when a service is requested, migrating threads allow the caller's thread to execute in the server's address space. This bypasses the scheduler code and avoids the cost of saving and restoring server state (registers, stacks). Ford's implementation reduces the RPC instruction count by 500% and increases the speed of RPC by about a factor of 2 when compared with Mach 3.0.

Interestingly, the migrating thread optimizations actually increase the machine's CPI. While Ford and Lepreau could not determine the exact cause of this increase, they believe that the higher percentage of load/store instructions in the migrating thread path, coupled with the fact the code was not carefully scheduled, could account for the decrease in architectural efficiency [Ford94]. The increase also underscores the interplay between software and hardware and the never ending need to evaluate performance at every level of the framework.

5.4.1.2 Data movement

Section 5.3.2 (p. 99) discussed how operating systems can reduce data movement by using virtual mapping techniques [Acetta86, Peterson90]. Mach 3.0 has extended this idea by mapping I/O devices into a user task's address space [Forin91]. This reduces the amount of user-kernel communication in some user-level servers.

Data movement can also be reduced through shared memory. Unlike virtual mapping, which occurs without the application's knowledge, shared memory is visible to

the application. In fact, the application must explicitly request that the operating system share memory between different address spaces. The X display server currently supports shared memory [Corbet91].

Shared memory is also very useful on machines with virtually-indexed caches because it removes most of the inconsistency problems associated with virtual mapping. It can, however, increase the number of TLB misses because the TLB will hold a separate entry for each address space. The HP Precision Architecture avoids this problem by supporting multiple PIDs per TLB entry [HP90].

5.4.2 OS design

Almost all of the issues discussed so far can be associated with some measurable cost. It may be difficult to determine the relative importance of the cost; it may be difficult to understand how much of the cost is due to the hardware vs. the software. But usually it is possible to isolate each component and determine some cost function.

This has not usually been true for high-level OS design issues. Combining all of the software components into sophisticated abstractions obscures the importance of various components and creates very complex and unforeseen interactions. This makes it difficult to determine the system's behavior and identify important performance issues.

Our results in Chapters 3 and 4 have shown that it is possible to correlate how high-level design decisions influence performance throughout the framework. However, this requires a careful understanding of each level's components and how they interact. The issues outlined in Section 5.2, Section 5.3 and Section 5.4.1 help architects and operating system designers understand these interactions. We now complete the discussion with two case studies.

5.4.2.1 Example 1 - File cache design

To improve file system performance, many operating systems reserve a portion of main memory for a file system cache. One version of the HP-UX operating system used the same file system structure (I-nodes) to manage both the disks and file cache. Braunstein et al. noticed that this design incurred substantial overhead[1] when managing linear address spaces such as RAM [Braunstein89]. In particular, seeks within a file were forced to traverse the I-node structure. To reduce this overhead, Braunstein et al. modified the OS, using the VM system to manage the file cache (the system was called XMF). This reduced the file cache's seek time, improving application performance by as much as 30%.

Because the new file cache design increased the amount of virtual space used by the system, Braunstein et al expected to see an increase in the TLB miss rate. When the TLB miss rate was measured, however, its performance had actually improved, accounting for 20% of the overall performance increase. At first, this result seemed counter-intuitive — increasing the size of the VM space should increase the number of PTEs a TLB must map. But this was not the case. The authors propose three possible reasons for the decrease in TLB miss rate.

- Mapping the file cache into the VM space reduced the instruction counts: fewer instructions, smaller working set, fewer TLB misses.
- The hardware TLB hashing algorithm interacted better with the "spread-out" references with XMF.
- The contiguous virtual address space references of XMF minimized the number of collisions on consecutive READs and WRITEs.

---

1. The overhead in terms of cycles is the same for both systems. But because RAM accesses are much quicker than disk accesses, the overhead as a percentage of total cost is much higher in the file cache.

This is an excellent example of how OS/architecture interactions often generate unexpected behavior, underscoring the fact that OS/architecture interactions play an integral part of system design and performance.

5.4.2.2 Example 2 - VM management

The wide range of virtual memory support found in today's architectures makes the OS's VM component one of the most difficult parts to port between machines. To ease this burden, OS designers have proposed separating the OS's virtual memory software into two distinct parts: the architecture-dependent code and the architecture independent code [Rashid88, Abrossimov89]. With a well designed VM system, the machine independent component can easily be moved between architectures by recompiling the source code, leaving only the machine-dependent code to be ported to each architectures's specific VM support. Mach and another micro-kernel operating system, Chorus, implement their virtual memory support using this type of structure [Rashid88, Abrossimov89].

Micro-benchmark measurements by Rasid et al. showed that Mach's VM system outperformed the native UNIX implementation [Rashid88] as measured by wall-clock time. However, their measurements also showed that Mach spent more CPU time (user+system) than either 4.3 BSD or SunOS 3.3 UNIX. While the wall-clock time is important to end-users, the amount of CPU time is important to system designers because it implies that the modularization of the VM system has an associated cost.

McRae investigated this cost using a hardware-based profiler [McRae93]. His results show that while Mach's VM performance was very good under micro-benchmarks, more complex tasks, such as creating a task (fork), did have performance problems. McRae traced much of the problem to the interface between the machine-independent and machine-depend portions of the VM system. McRae suggests that optimizing this path could improve overall performance.

5.5 Summary

Throughout this work, we have shown how OS/architecture interactions impact performance and have advocated continuing the analysis of this problem. In reviewing the literature, we realize that a significant number of performance studies have contributed to the area. However, there was no unifying model that could place each work's results within the context of the entire OS/architecture framework. This has made it difficult for designers to apply results outside of the specific systems.

We believe that the OS/architecture interactions framework is a step towards solving this problem. Comparing various works and integrating their results into the framework helps to build a more complete model which can be used to understand the impact each work has on other levels of the hierarchy. However, it is just a framework, not a complete picture of all system issues. Changes in architecture and OS systems can add too or alter the framework, changing the behavior of the system and the relative importance of system components.

CHAPTER 6
Where We Have Been and Where We are Going

6.1 Summary

Our work is an important contribution to the study of OS and architecture interactions. Techniques and tools similar to ours can be used by system designers to identify performance hot spots, possibly averting serious performance problems. During the course of this work, several announcements have shown that industry is responding to OS/architecture issues. The MIPS R4200 design improvements for WindowsNT and the IBM PowerPC's support for 68000 emulation are two examples. However, there are also examples where OS/architecture interactions have been overlooked. Apple was forced to delay the release of its PowerPC Macintosh notebook because the PowerPC 603 had very poor cache performance [MacWeek94]. Recently, IBM has announced the formation of a new group focused on analyzing performance issues in object-oriented programming, multimedia and microkernel operating systems [Clark94].

One of the limitations of any performance measurement study is that the results are intricately tied to the systems evaluated. Our results are no exception. A different TLB design could have withstood the increased stress presented by Mach. Better cache design with prefetching might have reduced the impact of Mach's larger working set. However, our results are based on real architectures and reflect real performance problems. Further, they provide insight into the causes of poor OS/architecture interactions, helping designers understand why design trade-offs improve performance. This is an important contribution

because while some designs may provide good performance, the reasons for the good performance can be unknown. This is especially true of OS/architecture interactions.

There are three results that are important to the types of systems we have evaluated. They are:

- Service decomposition and migration can degrade the performance of the TLB.
- Increasing code paths make the I-cache the most important architectural component.
- Changes in the architecture, such as better designed TLB systems and more sophisticated instruction fetching mechanisms can recapture much of the performance lost to OS and software trends.

Generalizing even further, we believe there are two basic issues that architects should consider. First, consider how the OS utilizes the caches. This is probably an obvious statement, but operating systems like user applications need carefully tuned caches. Second, provide a clean and efficient software/hardware interface. Most OS/architecture interface logic requires very little chip area and often is not in the critical path of the chip. Small enhancements such as additional registers for the OS or multiple interrupt vectors can really help OS implementation.

6.2 Where are we going

OS and architecture advances during the last several years have found their way into mainstream computing. Multi-issue processors, branch prediction, multi-level caches, multiple-API's, and microkernel (or at least multi-server) technologies are now available at your local computing store.

The design of future systems is not so clear. OS functionality will continue to grow, through technologies such as Taligent, and applications will increasingly rely on OS functionality. Architectures are also changing. HP and Intel have recently announced a

joint venture to develop a VLIW architecture. Other trends such as architecture emulation, Networks of Workstations (NOW), Personal Digital Assistants and Set-top Computing will also change the way we think about and use computing devices.

The increasingly important role operating systems are playing in computing is making OS performance critical to overall performance. We hope that as the number of OS/architecture performance studies grows, software and hardware designers will better understand which performance issues are the most important and how to design operating systems and architecture the work in concert, providing the best performance possible.

TERMINOLOGY

AFS	Andrew File System
continuation	A function that a thread should execute when it next runs.
cache2000	A trace-driven cache simulator developed by the MIPS Corporation.
CAM	Content Addressable Memory
CPI	Clock Cycles Per Instruction; CPI is one measure of the efficiency of the hardware.
CPI_{instr}	CPI due to instruction cache.
DMA	Direct Memory Access. Special hardware which performs I/O operations, allowing the CPU to perform other work.
IPC	interprocess communication
Mach 3.0	A micro-kernel version of the Mach operating system originally developed at Carnegie Mellon University
miss rate	misses / (misses + hits); typically used to describe the effectiveness of caches and TLBs
OS	Operating System
OSF/1	A version of Mach 2.5 originally developed by Carnegie Mellon University.
page table entry	a mapping from a task's virtual address space to the hardware's physical address space. Also used to hold protection and usage information (e.g. read only, valid).
PID	See process identifier.
pixie	A code annotation tool developed by MIPS Incorporated. pixie annotates a program so that when the program is run, the program will record the number of times each basic block was executed. It can also annotated the program to output the address of every instruction and data reference. This address stream can be used to drive architectural simulators like cache2000 or cheetah. Another tool, called pixstats, is used to perform static analysis on the basic block counts output by a pixified program.

pixstats	A static analysis program that determines the execution pattern of a program using a count of the number of times each basic block was executed.
process identifier	A unique number assigned to each process (task) in the system. In Mach there are also MIDs (Mach Identifiers) which
PTE	See Page Table Entry.
TLB	See Translation Lookaside Buffer
Translation Lookaside Buffer	An hardware structure used to cache recently used page table entries.
Ultrix	The type of Unix operating system used on the DECstation 3100. Derived from BSD 4.2.

BIBLIOGRAPHY

[Abrossimov89] Abrossimov, V. and Rozier, M. *Generic virtual memory management for operating system kernels*, In ACM, 123 - 136, 1989.

[Accetta86] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. *Mach: A new kernel foundation for UNIX development*, In Summer 1986 USENIX Conference, USENIX, 1986.

[Agarwal89] Agarwal, A. *Analysis of cache performance for operating systems and multiprogramming*. Stanford. 1989.

[Agarwal88] Agarwal, A., Hennessy, J. and Horowitz, M. *Cache performance of operating system and multiprogramming workloads. ACM Transactions on Computer Systems* **6** (Number 4): 393-431, 1988.

[Agarwal90] Agarwal, A. and Huffman, M. *Blocking: Exploiting spatial locality for trace compaction*, In The 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Boulder, CO, ACM, 48-57, 1990.

[Agarwal86] Agarwal, A., Sites, R. L. and Horowitz, M. *ATUM: A new technique for capturing address traces using microcode*, In Proceedings of the 13th International Symposium on Computer Architecture, Toyko, Japan, IEEE, 119-127, 1986.

[Alexander86] Alexander, C., Keshlear, W., Cooper, F. and Briggs, F. *Cache memory performance in a UNIX environment. Computer Architecture News* **14** : 14-70, 1986.

[Alexander85] Alexander, C. A., Keshlear, W. M. and Briggs, F. *Translation buffer performance in a UNIX environment. Computer Architecture News* **13** (5): 2-14, 1985.

[Alpert88] Alpert, D. and Flynn, M. *Performance trade-offs for microprocessor cache memories. IEEE Micro* (Aug): 44-54, 1988.

[AMD91] AMD. *Am29050 Microprocessor User's Manual.* Sunnyvale, CA, 1991.

[AMD93] AMD. *Am486 DX/DX2 Microprocessor Hardware Reference Manual.* Sunnyvale, CA, Advanced Micro Devices, Inc., 1993.

[Anderson91] Anderson, T. E., Levy, H. M., Bershad, B. N. and Lazowska, E. D. *The interaction of architecture and operating system design*, In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM, 108-119, 1991.

[Appel91] Appel, A. and Li, K. *Virtual memory primitives for user programs*, In The 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM, 96-107, 1991.

[Asprey93] Asprey, T., Averill, G. S., DeLano, E., Mason, R., Weiner, B. and Yetter, J. *Performance Features of the PA7100 Microprocessor. IEEE Micro* (June, 1993): 22 - 35, 1993.

[Baer87] Baer, J.-L. and Wang, W.-H. *Architectural choices for multi-level cache hierarchies. Proceedings of the 16th International Conference on Parallel Processing* : 258-261, 1987.

[Baer88] Baer, J.-L. and Wang, W.-H. *On the inclusion properties for multi-level cache hierarchies. The 15th Annual International Symposium on Computer Architecture* **16** (2): 73-80, 1988.

[Bala94] Bala, K., Kaashoek, M. F. and Weihl, W. E. *Software prefetching and caching for translation lookaside buffers*, In The First Symposium on Operating Systems Design and Implementation (OSDI), Monterey, CA, 243-253, 1994.

[Bershad90] Bershad, B. N., Anderson, T. E., Lazowska, E. D. and Levy, H. M. *Lightweight remote procedure call. ACM Transactions on Computer Systems* **8** (1): 37-55, 1990.

[Bershad92] Bershad, B. N. *The increasing irrelevance of IPC performance for microkernel-based operating systems*, In Micro-kernels and Other Kernel Architectures, Seattle, Washington, USENIX, 205-211, 1992.

[Bershad94a] Bershad, B. N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., PArdyak, P., Savage, S. and Sirer, E. G. *SPIN - An extensible microkernel for application-specific operating system services.* University of Washington. 94-03-03. 1994.

[Bershad94b] Bershad, B., Lee, D., Romer, T. and Chen, B. *Avoiding conflict misses dynamically in large direct-mapped caches*, In The Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, ACM Press (SIGOPS), 158-170, 1994.

[Biomation91] Biomation. *Biomation CLAS 4000 Application Note 4032.* Cupertino, CA, Biomation. 1991.

[Black89] Black, D. L., Rashid, R. F., Golub, D. B., Hill, C. R. and Baron, R. V. *Translation lookaside buffer consistency: A software approach*, In Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, ACM, 113-122, 1989.

[Black92] Black, D. L., Golub, D. B., Julin, D. P., Rashid, R. F., Draves, R. P., Dean, R. W., Forin, A., Barrera, J., Tokuda, H., Malan, G. and Bohman, D. *Microkernel operating system architecture and mach*, In Micro-kernels and Other Kernel Architectures, Seattle, Washington, USENIX, 11-30, 1992.

[Bomberger92] Bomberger, A., Hardy, N., Frantz, A. P., Landau, C. R., Frantz, W. S., Shapiro, J. S. and Hardy, A. C. *The KeyKOS Nanokernel Architecture*, In USENIX Micro-Kernels and Other Kernel Architectures, Seattle, Washington, USENIX, 95-112, 1992.

[Borg89] Borg, A., Kessler, R., Lazana, G. and Wall, D. *Long address traces from RISC machines: generation and analysis.* DEC Western Research Lab. 89/14. 1989.

[Borg90] Borg, A., Kessler, R. and Wall, D. *Generation and analysis of very long address traces*, In The 17th Annual International Symposium on Computer Architecture, IEEE, 1990.

[Braunstein89] Braunstein, A., Riley, M. and Wilkes, J. *Improving the efficiency of UNIX file buffer caches*, In Twelfth ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona, ACM, 71-82, 1989.

[Bray90] Bray, B., Lynch, W. and Flynn, M. J. *Page allocation to reduce access time of physical caches.* Stanford University, Computer Systems Laboratory. CSL-TR-90-454. 1990.

[Brunner91] Brunner, R. A. *VAX Architecture Reference Manual.* Digital Press, 1991.

[Budd91] Budd, T. *An Introduction to Object-Oriented Programming.* Addison-Wesley Publishing IBSN 0-201-54709-0, 1991.

[Busch85] Busch, J. R. and Kondoff, A. J. *Disc Caching in the System Processing Units of the HP 3000 Family of Computers. Hewlett-Packard Journal* **36** (2): 21 - 39, 1985.

[Calder94] Calder, B., Grunwald, D. and Zorn, B. *Quantifying behavioral differences between C and C++ programs.* The Department of Computer Science, University of Colorado. CU-CS-698-94. 1994.

[Campbell91] Campbell, R. H., Johnston, G. M., Madany, P. W. and Russo, V. F. *Principles of object-oriented operating system design.* 1991.

[Chao90] Chao, C., Mackey, M. and Sears, B. *Mach on a virtually addressed cache architecture*, In USENIX MACH Workshop Proceedings, Burlington, Vermont, USENIX, 31-51, 1990.

[Chase92] Chase, J. S., Levy, H. M., Baker-Harvey, M. and Lazowska, E. D. *How to use a 64-Bit virtual address space.* University of Washington. 92-03-02. 1992.

[Chen92] Chen, J. B., Borg, A. and Jouppi, N. P. *A simulation based study of TLB performance*, In The 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, IEEE, 114-123, 1992.

[Chen93a] Chen, B. *Software methods for system address tracing (tech report).* CMU-CS-93-188. 1993.

[Chen93b] Chen, B. *Software methods for system address tracing*, In Proceedings of the Fourth Workshop on Workstation Operating Systems, Napa, California, 1993.

[Chen93c] Chen, B. and Bershad, B. *The impact of operating system structure on memory system performance*, In Proc. 14th Symposium on Operating System Principles, 1993.

[Chen94a] Chen, B. *Memory behavior of an X11 window system*, In USENIX Winter 1994 Technical Conference, 1994.

[Chen94b] Chen, B., Wall, D. and Borg, A. *Software methods for system address tracing: implementation and validation.* Carnegie-Mellon University, DEC Western Research Lab, DEC Network Systems Laboratory. 1994.

[Cheriton88] Cheriton, D. *The vmp multiprocessor: Initial experience, refinements and performance evaluation*, In Proc. of the 14th Annual Sumposium on Computer Architecture, 1988.

[Cheriton84] Cheriton, D. R. *The V kernel: A software base for distributed systems. IEEE Software* **1** (2): 19-42, 1984.

[Chu85] Chu, C. *MILS: Mips instruction level simulator.* 1985.

[Clark83] Clark, D. *Cache performance in the VAX-11/780. ACM Transactions on Computer Systems* **1** : 24-37, 1983.

[Clark85a] Clark, D. W., Bannon, P. J. and Keller, J. B. *Measuring VAX 8800 Performance with a Histogram Hardware Monitor,* In The 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, IEEE, 176-185, 1985.

[Clark88] Clark, D. W. and Emer, J. S. *Performance of the VAX-11/780 translation buffer: Simulation and measurement. ACM Transactions on Computer Systems* **3** (1): 31-62, 1985.

[Clark94] Clark, M. *IBM preps PowerPC tweaks.* Electronic Engineering Times. 16, 1994.

[Corbet91] Corbet, J. *MIT-SHM The MIT shared memory extension.* Massaschusetts Institute of Technology. 1991.

[Covington88] Covington, R. C., Madala, S., Mehta, V., Jump, J. R. and Sinclair, J. B. *The Rice parallel processing testbed,* In SIGMETRICS, ACM, 4-11, 1988.

[Custer93] Custer, H. *Inside Windows NT.* Redmond, Washington, Microsoft Press, 1993.

[Cvetanovic94] Cvetanovic, Z. and Bhandarkar, D. *Characterization of Alpha AXP performance using TP and SPEC Workloads,* In The 21st Annual International Symposium on Computer Architecture, Chicago, Ill., IEEE, 1994.

[Cypress90] Cypress. *SPARC RISC Users Guide.* ROSS Technology Inc, A Cypress Semiconductor Company, 1990.

[Dean91] Dean, R. W. and Armand, F. *Data movement in kernelized systems,* In Micro-kernels and Other Kernel Architectures, Seattle, Washington, USENIX, 243-261, 1991.

[DeMoney86] DeMoney, M., Moore, J. and Mashey, J. *Operating system support on a RISC,* In COMPCON, 138-143, 1986.

[Digital81] Digital. *VAX Architecture Handbook.* Digital, 1981.

[Digital86] Digital. *VAX architecture handbook.* Bedford, MA, Digital Equipment Corporation, 1986.

[Digital92] Digital. *Alpha Architecture Handbook.* USA, Digital Equipment Corporation, 1992.

[Draves91] Draves, R. P., Bershad, B. N., Rashid, R. F. and Dean, R. W. *Using continuations to implement thread management and communication in operating systems*, In Proceedings of the 13th ACM Symposium on Operating Systems Principles, 122-136, 1991.

[Druschel92] Druschel, P., Peterson, L. L. and Hutchinson, N. C. *Beyond micro-kernel design: Decoupling modularity and protection in Lipto*, In Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, IEEE, 512-520, 1992.

[Eickemeyer88] Eickemeyer, R. and Patel, J. *Performance evaluation of on-chip register and cache organizations*, In Proc. 15th Annual Symposium on Computer Architecture, Honolulu, Hawaii, 64-72, 1988.

[Emer84] Emer, J. and Clark, D. *A characterization of processor performance in the VAX-11/780*, In The 11th Annual Symposium on Computer Architecture, Ann Arbor, MI, IEEE, 301-309, 1984.

[Fall93] Fall, K. and Pasquale, J. *Exploiting in-kernel data paths to improve I/O throughput and cpu availability*, In 1993 Winter USENIX, San Diego, CA, USENIX, 327-333, 1993.

[Farrens89] Farrens, M. and Pleszkun, A. *Improving performance of small on-chip instruction caches*, In The 16th Annual International Symposium on Computer Architecture, ACM, 234-241, 1989.

[Flanagan92a] Flanagan, K., Grimsrud, K., Archibald, J. and Nelson, B. *BACH: BYU address collection hardware.* Brigham Young University. TR-A150-92.1. 1992.

[Flanagan92b] Flanagan, J. K., Nelson, B. E., Archibald, J. K. and Grimsrud, K. *BACH: BYU address collection hardware, the collection of complete traces*, In The 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, 128-137, 1992.

[Flanagan93a] Flanagan, J. K., Nelson, B. E. and Archibald, J. K. *The inaccuracy of trace-driven simulation using incomplete trace data.* Brigham Young University. 1993.

[Flanagan93b] Flanagan, J. K. *A new methodology for accurate trace collection and its application to memory heirarchy performance modeling.* Brigham Young University. 1993.

[Flanagan93c] Flanagan, J. K., Nelson, B. E., Archibald, J. K. and Grimsrud, K. *Incomplete trace data and trace-driven simulation*, In The International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS, 203-209, 1993.

[Flanagan94] Flanagan, J. K. *Personal Communication.* 1994.

[Ford94] Ford, B. and Lepreau, J. *Evolving Mach 3.0 to a migrating thread model.* University of Utah. 1994.

[Forin91] Forin, A., Golub, D. and Bershad, B. *An I/O system for Mach 3.0*, In USENIX Mach Symposium, Monterey, CA, USENIX, 163-176, 1991.

[Fuentes93] Fuentes, C. *Hardware support for operating systems.* The University of Michigan. 1993.

[Gee93] Gee, J., Hill, M., Pnevmatikatos, D. and Smith, A. J. *Cache Performance of the SPEC92 Benchmark Suite. IEEE Micro* (August): 17-27, 1993.

[Ginsberg93] Ginsberg, M., BAron, R. V. and Bershad, B. N. *Using the mach communication primitives in X11.* Carnegie Mellon University. 1993.

[Golub90] Golub, D., Dean, R., Forin, A. and Rashid, R. *Unix as an application program*, In Proceedings of the USENIX Summer Conference, USENIX, 1990.

[Golub91] Golub, D. and Draves, R. *Moving the default memory manager out of the Mach kernel*, In USENIX Mach Symposium, Monterey, CA, USENIX, 177-188, 1991.

[Goodman83] Goodman, J. *Using cache memory to reduce processor memory traffic*, In Tenth International Symposium on Computer Architecture, Stockholm, Sweden, 124-131, 1983.

[Goodman86] Goodman, J. and Hsu, W.-C. *On the use of registers vs. cache to minimize memory traffic*, In Proc. 13th International Symposium on Computer Architecture, 375-383, 1986.

[Goscinski91] Goscinski, A. *Distributed Operating Systems.* Singapore, Addison-Wesley Publishers Ltd., 1991.

[Grimsrud92] Grimsrud, K., Archibald, J., Frost, R., Nelson, B. and Flanagan, K. *Estimation of simulation error due to trace inaccuracies*, In The 26th Asilomar Conference on Signals, Systems, and Computers, 1992.

[Grimsrud93a] Grimsrud, K., Archibald, J., Ripley, M., Flanagan, K. and Nelson, B. *BACH: A hardware monitor for tracing microprocessor-based systems*, In Microprocessors and Microsystems, to appear, 1993.

[Grimsrud93b] Grimsrud, K. S. *Quantifying Locality.* Brigham Young University. 1993.

[Grimsrud93c] Grimsrud, K. *Address translation of BACH i486 traces.* Brigham Young University. 1993.

[Hammerstrom77] Hammerstrom, D. and Davidson, E. *Information content of CPU memory referencing behavior*, In Proceedings of the Fourth International Symposium on Computer Architecture, 184-192, 1977.

[Happel92] Happel, L. P. and Jayasumana, A. P. *Performance of a RISC machine with two-level caches. IEE Proceedings-E* **139** (3): 221-229, 1992.

[Hennessy90] Hennessy, J. L. and Patterson, D. A. *Computer Architecture A Quantitative Approach.* San Mateo, Morgan Kaufmann, 1990.

[Henry83] Henry, R. R. *VAX address and instruction traces.* University of California at Berkeley. 1983.

[HP90] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Hewlett-Packard, Inc., 1990.

[HP91] Hewlett-Packard. *Test and Measurement Catalog.* Santa Clara, CA, Marketing Communications, 1991.

[Hill87] Hill, M. *Aspects of cache memory and instruction buffer performance.* The University of California at Berkeley. 1987.

[Hill89] Hill, M. and Smith, A. *Evaluating associativity in CPU caches. IEEE Transactions on Computers* **38** (12): 1612-1630, 1989.

[Hill84] Hill, M. and Smith, A. J. *Experimental evaluation of on-chip microprocessor cache memories*, In 11th Annual International Symposium on Computer Architecture, Ann Arbor, Michigan, 158-166, 1984.

[Huck93] Huck, J. and Hays, J. *Architectural support for translation table management in large address space machines*, In The 20th Annual International Symposium on Computer Architecture, San Diego, California, IEEE, 39-50, 1993.

[Hwu89] Hwu, W.-m. and Chang, P. *Achieving high instruction cache performance with an optimizing compiler*, In The 16th International Symposium on Computer Architecture, Jerusalem, Isreal, IEEE Computer Society Press (ACM SIGARCH), 242-251, 1989.

[IBM90] IBM. *IBM RISC System/6000 Technology.* Austin , TX, IBM, 1990.

[IBM93] IBM. *PowerPC 601.* IBM Microelectronics and Motorola, 1993.

[Inouye] Inouye, J., Konuru, R., Walpole, J. and Sears, B. *The effects of virtually addressed caches on virtual memory performance*, In 14 - 29,

[Intel90] Intel. *i860 64-bit Microprocessors Programmer's Manual.* Santa Clara, CA, Intel Corporation, 1990.

[ISCA92] ISCA92. *The 19th Annual International Symposium on Computer Architecture*, In ISCA, Gold Coast, Australia, ACM SIGARCH and IEEE Computer Society, 1-424, 1992.

[ISCA93] ISCA93. *The 20th Annual International Symposium on Computer Architecture*, In ISCA, San Diego, California, ACM SIGARCH and IEEE Computer Society, 1-360, 1993.

[ISCA94] ISCA94. *The 21st Annual International Symposium on Computer Architecture*, In ISCA, Chicago, Illinois, ACM SIGARCH and IEEE Computer Society, 1-394, 1994.

[Jouppi90] Jouppi, N. *Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers*, In The 17th Annual International Symposium on Computer Architecture, Seattle, WA, IEEE Computer Society Press (ACM SIGARCH), 364-373, 1990.

[Jouppi94] Jouppi, N. and Wilton, S. *Tradeoffs in two-level on-chip caching*, In The 21st Annual International Symposium on Computer Architecture, Chicago, IL, IEEE Computer Society Press, 34-45, 1994.

[Kane92] Kane, G. and Heinrich, J. *MIPS RISC Architecture.* Prentice-Hall, Inc., 1992.

[Kay93] Kay, J. and Pasquale, J. *Measurement, analysis and implementation of UDP/IP throughput for the DECstation 5000*, In 1993 Winter USENIX, San Diego, CA, USENIX, 259-258, 1993.

[Kessler91] Kessler, R. *Analysis of multi-megabyte secondary CPU cache memories*. University of Wisconsin-Madison. 1991.

[Kessler92] Kessler, R. and Hill, M. *Page placement algorithms for large real-indexed caches. ACM Transaction on Computer Systems* **10** (4): 338-359, 1992.

[Khalidi93] Khalidi, Y. A. and Nelson, M. N. *An implementation of UNIX on an Object-oriented operating system*, In Proceedings of Winter '93 USENIX Conference, San Diego, California, USENIX, 469-480, 1993.

[Koch94] Koch, P. *Emulating the 68040 in the PowerPC Macintosh*, In Microprocessor Forum, San Francisco, CA, 1994.

[Koldinger92] Koldinger, E. J., Chase, J. S. and Eggers, S. J. *Architectural support for single address space operating systems*, In Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, ACM, 175-186, 1992.

[Kuck88] Kuck, D. and al., e. *The perfect club benchmarks: effective performance evaluation of supercomputers*. The University of Illinois. CSRD Report. 1988.

[Lacy88] Lacy, F. *An address trace generator for trace-driven simulation of shared memory multiprocessors*. University of California at Berkeley. Technical Report UCB/CSD 88/407. 1988.

[Laha88] Laha, S., Patel, J. and Iyer, R. *Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Transactions on Computers* **37** (11): 1325-1336, 1988.

[Larus90] Larus, J. R. *Abstract Execution: A technique for efficiently tracing programs*. University of Wisconsin-Madison. 1990.

[Larus91] Larus, J. *SPIM S20: A MIPS R2000 Simulator*. University of Washington. Revision 9. 1991.

[Larus93] Larus, J. R. *Efficient program tracing. IEEE Computer* **May, 1993** : 52-60, 1993.

[Lee94] Lee, C.-C. *A case study of a hardware-managed TLB in a multi-tasking environment*. The University of Michigan. 1994.

[Levy80] Levy, H. M. and Jr., R. H. E. *Computer Programming and Architecture: The VAX-11*. Bedford, Mass., Digital Press, 1980.

[Lynch93] Lynch, W. L. *The Interaction of Virtual Memory and Cache Memory*. Stanford University. CSL-TR-93-587. 1993.

[MacWeek94] MacWeek. *Apple holds up 603 for cache*. MacWeek. **21:** 1, 1994.

[Malan91] Malan, G., Rashid, R., Golub, D. and Baron, R. *DOS as a Mach 3.0 application*, In USENIX Mach Symposium, USENIX, 27-40, 1991.

[Martonosi92] Martonosi, M., Gupta, A. and Anderson, T. *MemSpy: Analyzing memory system bottlenecks in programs*, In SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, ACM, 1992.

[Martonosi93] Martonosi, M., Gupta, A. and Anderson, T. *Effectiveness of trace sampling for performance debugging tools*, In SIGMETRICS, Santa Clara, California, ACM, 248-259, 1993.

[Martonosi94] Martonosi, M. *Analyzing and tuning memory performance in sequential and parallel programs*. Stanford University. 1994.

[Mattson70] Mattson, R. L., Gecsei, J., Slutz, D. R. and Traiger, I. L. *Evaluation Techniques for Storage Hierarchies. IBM Systems Journal* **9** (2): 78-117, 1970.

[May87] May, C. *Mimic: A fast S/370 simulator*, In Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices, St. Paul, Minnesota, ACM, 1-13, 1987.

[Maynard94] Maynard, A. M., Donnelly, C. and Olszewski, B. *Contrasting characteristics and cache performance of technical and multi-user commercial workloads*, In The Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, ACM Press (SIGOPS), 145-156, 1994.

[McCormack91] McCormack, J. *Writing Fast X Servers for Dumb Color Frame Buffers*. Digital Western Research Laboratory. 1991.

[McFarling89] McFarling, S. *Program optimization for instruction caches*, In The Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, ACM (SIGARCH), 183-191, 1989.

[McKusick84] McKusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S. *A fast file system for UNIX. ACM Transactions on Computer Systems* **2** (3): 181-197, 1984.

[McRae93] McRae, A. *Hardware profiling of kernels*, In 1993 Winter USENIX, San Diego, CA, USENIX, 375-386, 1993.

[Milandre75] Milandre, G. and Mikkor, R. *VS2-R2 experience at the University of Toronto computer centre*, In SHARE44 Proceedings, Los Angelas, California, 1887- 1895, 1975.

[Milenkovic90] Milenkovic, M. *Microprocessor Memory Management Units. IEEE Micro* **10** (2): 70-85, 1990.

[MIPS88] MIPS. *RISCompiler Languages Programmer's Guide*. MIPS, 1988.

[Mogul91] Mogul, J. C. and Borg, A. *The effect of context switches on cache performance*, In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM, 75-84, 1991.

[Motorola90a] Motorola. *MC88200 Cache/Memory Management Unit User's Manual*. Prentice Hall, 1990.

[Motorola90b] Motorola. *MC88100 RISC Microprocessor User's Manual*. Englewood Cliffs, NJ, Prentice Hall, 1990.

[Motorola93] Motorola. *PowerPC 601 RISC Microprocessor Users' Manual*. Motorola, Inc., 1993.

[MReport92] MReport. Sebastopol, CA, MicroDesign Resources, 1992.

[MReport93] MReport. Sebastopol, CA, MicroDesign Resources, 1993.

[MReport94] MReport. Sebastopol, CA, MicroDesign Resources, 1994.

[Mulder91] Mulder, J., Quach, N. and Flynn, M. *An area model for on-chip memories and its application. IEEE Journal of Solid-State Circuits* **26** (2): 98-106, 1991.

[Nagle92] Nagle, D., Uhlig, R. and Mudge, T. *Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures*. The University of Michigan. CSE-TR-147-92. 1992.

[Nagle93] Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T. and Brown, R. *Design tradeoffs for software-managed TLBs*, In The 20th Annual International Symposium on Computer Architecture, San Diego, California, IEEE, 27-38, 1993.

[Nagle94] Nagle, D., Uhlig, R., Mudge, T. and Sechrest, S. *Optimal Allocation of On-chip Memory for Multiple-API Operating Systems*, In The 21st International Symposium on Computer Architecture, Chicago, IL, 1994.

[Nelson90] Nelson, M. N. *Virtual memory vs. the file system.* Digital Wester Research Laboratory. 1990.

[Nielson91] Nielson, R. *DOS on the dock.* NeXT World. 50-51, 1991.

[Olukotun91] Olukotun, O. A., Mudge, T. N. and Brown, R. B. *Implementing a cache for a high-performance GaAs microprocessor*, In Proc. 18th Annual International Symposium on Computer Architecture, Toronto, Canada, 138-147, 1991.

Olukotun92] Olukotun, K., Mudge, T. and Brown, R. *Performance optimization of pipelined primary caches*, In The 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, IEEE, 181-190, 1992.

[Ousterhout89] Ousterhout, J. *Why aren't operating systems getting faster as fast as hardware. WRL Technical Note* (TN-11): 1989.

[Ousterhout94] Ousterhout, J. K. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, 1994.

[Palcharla94] Palcharla, S. and Kessler, R. E. *Evaluating stream buffers as a secondary cache replacement*, In The 21st Annual International Symposium on Computer Architecture, Chicago, Illinois, IEEE, 24-33, 1994.

[Patel92] Patel, K., Smith, B. C. and Rowe, L. A. *Performance of a Software MPEG Video Decoder.* University of California, Berkeley. 1992.

[Patterson80] Patterson, D. A. and Sequin, C. H. *Design considerations for single-chip computers of the future. IEEE Transactions on Computers* **C-29** (2): 108-116, 1980.

[Peterson90] Peterson, L., Hutchinson, N., O'Malley, S. and Rao, H. *The x-kernel: A platform for accessing internet resources. IEEE Computer* **23** (5): 23-33, 1990.

[Pierce94a] Pierce, J. and Mudge, T. *IDtrace - A tracing tool for i486 simulation (technical report).* University of Michigan. CSE-TR-203-94. 1994.

[Pierce94b] Pierce, J. and Mudge, T. *IDtrace - A tracing tool for i486 simulation (extended abstract)*, In The International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS), 419-420, 1994.

[Pierce94c] Pierce, J. and Mudge, T. *The effect of speculative execution on cache performance*, In The International Parallel Processing Symposium, Cancun, Mexico, April, 1994.

[Prieve74] Prieve, B. G. *A page partition replacement algorithm.* University of California at Berkeley. 1974.

[Przybylski89] Przybylski, S., Horowitz, M. and Hennessy, J. *Characteristics of performance-optimal multi-level cache hierarchies*, In Proc. 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, 114-121, 1989.

[Przybylski90] Przybylski, S. *The performance impact of block sizes and fetching strategies*, In Proceedings of the 16th Annual International Symposium on Computer Architecture, Seattle, WA, IEEE, 160-169, 1990.

[Puzak85] Puzak, T. *Analysis of cache replacement algorithms.* University of Massachusetts. 1985.

[Rashid88] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. IEEE Transactions on Computers* **37** (8): 896-908, 1988.

[Reinhardt93] Reinhardt, S., Hill, M., Larus, J., Lebeck, A., Lewis, J. and Wood, D. *The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers*, In SIGMETRICS 93 (Special Issue of Performance Evaluation Review), Santa Clara, CA, ACM, 48-60, 1993.

[Romer94] Romer, T. H., Lee, D., Bershad, B. N. and Chen, J. B. *Dynamic page mapping policies for cache conflict resolution on standard hardware*, In The First Symposium on Operating Systems Design and Implementation (OSDI), Monterey, CA, 255-266, 1994.

[Rozier92] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaise, C., Langlois, S., Leonard, P. and Neuhauser, W. *Overview of the Chorus distributed operating system*, In Micro-kernels and Other Kernel Architectures, Seattle, Washington, USENIX, 39-69, 1992.

[Samples89] Samples, A. *Mache: no-loss trace compaction*, In Proceedings of 1989 ACM Sigmetrics and Performance '89 International Conference on Measurement and Modeling of Computer Systems, ACM, 89-97, 1989.

[Satyanarayanan90] Satyanarayanan, M. *Scalable, secure, and highly available distributed file access. IEEE Computer* **23** (5): 9-21, 1990.

[Scheifler86] Scheifler, R. and Gettys, J. *The X window system. ACM Transactions on Graphics* **5** (2): 79-109, 1986.

[Short88] Short, R. and Levy, H. *A simulation study of two-level caches*, In Proc. 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, 81-88, 1988.

[Sites88] Sites, R. L. and Agarwal, A. *Multiprocessor cache analysis with ATUM*, In The 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, IEEE, 186-195, 1988.

[Sites92] Sites, R., Chernoff, A., Kirk, M., Marks, M. and Robinson, S. *Binary translation. Digital Technical Journal* **4** (4): 137-152, 1992.

[Smith77] Smith, A. J. *Two methods for the efficient analysis of memory address trace data. IEEE Transactions on Software Engineering* **SE-3** (1): 94-101, 1977.

[Smith78] Smith, A. J. *Sequential program prefetching in memory hierarchies. IEEE Computer* **11** (12): 7-21, 1978.

[Smith82] Smith, A. J. *Cache Memories. Computing Surveys* **14** (3): 473-530, 1982.

[Smith85] Smith, A. J. *Cache evaluation and the impact on workload choice*, In 12th International Symposium on Computer Architecture, Boston, Mass., IEEE, 64-73, 1985.

[Smith86] Smith, A. *Bibliography and readings on CPU cache memories and related topics. Computer Architecture News* **14** : 22-42, 1986.

[Smith89] Smith, M. D., Johnson, M. and Horowitz, M. A. *Limits on Multiple Instruction Issue*, In Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, ACM, 290-302, 1989.

[Smith91] Smith, M. D. *Tracing with pixie*. Stanford University, Stanford, CA. 1991.

[Smith92] Smith, J. E. and Hsu, W.-C. *Prefetching in supercomputer instruction caches*, In Supercomputing '92, 588-597, 1992.

[Smith94] Smith, J. E. and Weiss, S. *PowerPC 601 and Alpha 21064: A tale of two RISCs. IEEE Computer* **27** (6): 46-58, 1994.

[SPEC91] SPEC. *The SPEC Benchmark Suite*. SPEC Newsletter. **3:** 3-4, 1991.

[SPEC93] SPEC. *SPEC: A five year retrospective. The SPEC Newsletter* **5** (4): 1-4, 1993.

[Srivastava94] Srivastava, A. and Eustace, A. *ATOM: A system for building customized program analysis tools.* DEC Western Research Lab. TN-41. 1994.

[Stephens91] Stephens, C., Cogswell, B., Heinlein, J., Palmer, G. and Shen, J. *Instruction level profiling and evaluation of the IBM RS/6000*, In The 18th Annual International Symposium on Computer Architecture, Toronto, Canada, ACM, 180-189, 1991.

[Stodolsky93] Stodolsky, D., Chen, J. B. and Bershad, B. N. *Fast interrupt priority management in operating system kernels.* Carnegie Mellon University. CMU-CS-93-152. 1993.

[Stolarchuk92] Stolarchuk, M. T. *Faster AFS.* Center for Information Technology Integration. 92-3. 1992.

[Stone93] Stone, H. *High-performance Computer Architecture.* Reading, Massachusetts, Addison-Wesley, 1993.

[Stunkel89] Stunkel, C. and Fuchs, W. *TRAPEDS: producing traces for multicomputers via execution-driven simulation*, In Proceedings of the 1989 ACM Sigmetrics and Performance '89 International Conference on Measurement and Modeling of Computer Systems, Berkeley, CA, ACM, 70-78, 1989.

[Stunkel91] Stunkel, C., Janssens, B. and Fuchs, W. K. *Collecting address traces from parallel computers*, In The 24th Annual Hawaii Internation Conference on System Sciences, Hawaii, 373-383, 1991.

[Sugumar93] Sugumar, R. *Multi-configuration simulation algorithms for the evaluation of computer designs.* University of Michigan. 1993.

[Talluri92] Talluri, M., Kong, S., Hill, M. D. and Patterson, D. A. *Tradeoffs in supporting two page sizes*, In The 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia, 415-424, 1992.

[Talluri94] Talluri, M. and Hill, M. *Surpassing the TLB Performance of Superpages with Less Operating System Support*, In ASPLOS-VI, San Jose, CA, ACM, In this proceedings, 1994.

[Taylor90] Taylor, G., Davies, P. and Farmwald, M. *The TLB slice - A low-cost high-speed address translation mechanism*, In The 17th Annual International Symposium on Computer Architecture, 355-363, 1990.

[Tektronix94] Tektronix. *Test and Mesurement Product Catalog*. Wilsonville, OR, 1994.

[Thekkath94] Thekkath, C. and Levy, H. *Hardware and software support for efficient exception handling*, In The 6th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, ACM Press, 110-119, 1994.

[Torrellas92] Torrellas, J., Gupta, A. and Hennessy, J. *Characterizing the caching and synchronization performance of multiprocessor operating system*, In Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, ADM, 162-174, 1992.

[Torrellas95] Torrellas, J., Xia, C. and Daigle, R. *Optimizing instruction cache performance for operating system intensive workloads*, In The 1st International Syposium on High-Performance Computer Architecture (HPCA), Raleigh, North Carolina, to appear, 1995.

[Touma92] Touma, W. R. *The Dynamics of the Computer Industry.* University of Texas at Austin. 1993.

[Uhlig92] Uhlig, R., Nagle, D., Mudge, T. and Sechrest, S. *Software TLB management in OSF/1 and Mach 3.0.* University of Michigan. CSE-TR-156-93. 1992.

[Uhlig94a] Uhlig, R. *Kernel-based Memory Simulation (Extended Abstract)*, In SIGMETRICS, Nashville, TN, University of Michigan, 286-287, 1994.

[Uhlig94b] Uhlig, R., Nagle, D., Stanley, T., Sechrest, S., Mudge, T. and Brown, R. *Design tradeoffs for software-managed TLBs. ACM Transactions on Computer Systems* (Fall): To appear, 1994.

[Uhlig94c] Uhlig, R., Nagle, D., Mudge, T. and Sechrest, S. *Trap-driven simulation with Tapeworm II*, In The Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, ACM Press (SIGARCH), 132-144, 1994.

[Uhlig95] Uhlig, R., *Trap-driven simulation*, Dissertation, University of Michigan, Ann Arbor, Michigan, 1995.

[Upton94] Upton, M. D. *Architectural trade-offs in a latency tolerant gallium arsenide microprocessor.* The University of Michigan. 1994.

[Veenstra94] Veenstra, J. and Fowler, R. *MINT: A front end for efficient simulation of shared-memory multiprocessors*, In The 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication systems (MASCOTS), 201-207, 1994.

[Wada92] Wada, T., Rajan, S. and Przybylski, S. *An analytical access time model for on-chip cache memories. IEEE Journal of Solid-State Circuits* **27** (8): 1147-1156, 1992.

[Wall89] Wall, D. *Link-time code modification.* DEC Western Research Lab. 89/17. 1989.

[Wall92] Wall, D. *Systems for late code modification.* DEC Western Research Lab. 92/3. 1992.

[Wang90] Wang, W.-H. and Baer, J.-L. *Efficient trace-driven simulation methods for cache performance analysis*, In The 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Boulder, CO, ACM, 27-36, 1990.

[Wang89] Wang, W.-H., Baer, J.-L. and Levy, H. *Organization and performance of a two-level virtual-real cache hierarchy*, In The 16th Annual International Symposium on Computer Archtecture, Jerusalem, Isreal, IEEE Computer Society Press (ACM SIGARCH), 140-148, 1989.

[Welch91] Welch, B. *The file system belongs in the kernel*, In USENIX Mach Symposium Proceedings, Monterey, California, USENIX, 233-249, 1991.

[Wheeler92] Wheeler, B. and Bershad, B. N. *Consitency management for virtually indexed caches*, In Fifth Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, 1992.

[Wiecek82] Wiecek, C. A. *A case study of VAX-11 instruction set usage for compiler execution*, In Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, 177-184, 1982.

[Wiecek92] Wiecek, C. A., Kaler, C. G., Fiorelli, S., Davenport, W. C. and Chen, R. C. *A Model and Prototype of VMS Using the Mach 3.0 Kernel*, In USENIX Micro-kernels and Other Kernel Architectures Workshop, Seattle, Washington, USENIX, 187-203, 1992.

[Wilkes92] Wilkes, J. and Sears, B. *A comparison of protection lookaside buffers and the PA-RISC protection architecture.* HP Laboratories. HPL-92-55. 1992.

[Wilton94] Wilton, S. and Jouppi, N. *An enhanced access and cycle time model for on-chip caches.* DEC Western Research Lab. 93/5. 1994.

[Winsor89] Winsor, D. *Bus and cache memory organizations for multi-processors.* The University of Michigan. 1989.

[Wood86] Wood, D., Eggers, S., Gibson, G., Hill, M., Pendleton, J., Ritchie, S., Taylor, G., Katz, R. and Patterson, D. *An in-cache address translation mechanism*, In The 13th Annual Symposium on Computer Architecture, IEEE Computer Society Press, 358-365, 1986.

[Wood91] Wood, D., Hill, M. and Kessler, R. *A model for estimating trace-sampled miss ratios*, In ACM SIGMETRICS, 1991.