To Appear in *The Fourteenth Symposium on Operating System Principles.*

The Impact of Operating System Structure on Memory System Performance

J. Bradley Chen
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Brian N. Bershad
Department of Computer Science
and Engineering
University of Washington
Seattle, WA 98195

Abstract

In this paper we evaluate the memory system behavior of two distinctly different implementations of the UNIX operating system: DEC's Ultrix, a monolithic system, and Mach 3.0 with CMU's UNIX server, a microkernel-based system. In our evaluation we use combined system and user memory reference traces of thirteen industry-standard workloads. We show that the microkernel-based system executes substantially more non-idle system instructions for an equivalent workload than the monolithic system. Furthermore, the average instruction for programs running on Mach has a higher cost, in terms of memory cycles per instruction, than on Ultrix. In the context of our traces, we explore a number of popular assertions about the memory system behavior of modern operating systems, paying special attention to the effect that Mach's microkernel architecture has on system performance. Our results indicate that many, but not all of the assertions are true, and that a few, while true, have only negligible impact on real system performance.

This research was sponsored in part by the Advanced Research Projects Agency, Information Science and Technology Office, ARPA Order Nos. 7330 and 7597, the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, and Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Digital Equipment Corporation or the U.S. Government.

1. Introduction

In this paper we quantitatively evaluate the memory system behavior of two different implementations of the UNIX operating system. One system, DEC's Ultrix, has a monolithic structure. The other, Mach 3.0 with CMU's UNIX server [1, 21], has a microkernel structure. Both systems are derived from 4.2 BSD UNIX and share a nearly identical application programming interface, as well as large amounts of code. We explore these two systems within the framework of seven popular assertions about the memory reference behavior of modern operating systems. These assertions, listed in Table 1-1, arise from past experiences [16], extrapolated microbenchmarks [9, 31], and extensive measurements of real systems running real programs [3, 4, 5, 14, 15, 28, 35, 36]. Our evaluation relies on combined system and user memory reference traces generated through software instrumentation of the systems running a broad selection of workloads.

Previous trace-based studies have focused on variations in memory system structure [2, 3, 4, 10, 13, 28, 32], multiprocessors and multiprocessor workloads [35, 36], or subcomponents of the memory system [30]. In contrast, our goal is to explore the impact of operating system structure on the performance of a complete uniprocessor memory system. The rest of this paper is structured as follows. In Section 2 we describe our trace methodology and present a broad summary of our measurements. In Section 3 we discuss the major differences in system behavior and performance between Mach and Ultrix. In Section 4 we evaluate the monolithic and microkernel implementations in the context of the assertions in Table 1-1. Finally, in Section 5 we summarize our results.

2. Trace overview

We measured the behavior of Ultrix and Mach running the thirteen industry-standard workloads described in Table 2-2. Each program and operating system was instrumented with *epoxie* [10, 37], which is a program that rewrites assembly code to record a complete address trace

Assertion	Implication
1. The operating system has less instruction and data locality than user programs [14, 15].	The operating system isn't getting faster as fast as user programs.
2. System execution is more dependent on instruction cache behavior than is user execution [35].	A balanced cache system for user programs may not be balanced for the system.
3. Collisions between user and system references lead to significant performance degradation in the memory system (cache and TLB) [30, 35, 36].	A split user/system cache could improve performance.
4. Self-interference is a problem in system instruction reference streams [28, 35].	Increased cache associativity and/or the use of text placement tools could improve performance.
5. System block memory operations are responsible for a large percentage of memory system reference costs [31, 35].	Programs that incur many block memory operations will run more slowly than expected.
6. Write buffers are less effective for system (as opposed to user) reference streams [5, 18].	A write buffer adequate for user code may not be adequate for system code.
7. Virtual page mapping strategies can have significant impact on cache performance [25, 29].	Systems should support a flexible page mapping interface, and should avoid default strategies that are prone to pathological behavior.

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

of instruction and memory references. Traces are accurately interleaved both within a single context and across user and system contexts. Traced addresses are corrected to reflect those of the original and not the traced instruction stream. The same traced binaries are run on both Ultrix and Mach.

We collected our traces on a DECstation 5000/200 system,[1] running Ultrix and Mach 3.0 with CMU's UNIX server.[2] We ran the programs one at a time in single-user mode. The only activity was due to the program itself, the kernel, and in the case of Mach, the user-level operating system server. Some of the experiments described in this paper required that we run the same program several times against different simulated memory systems. To ensure consistency from run to run, the system was rebooted before every experiment.

The trace is fed into a simulation of the DS5000/200 memory system using the parameters shown in Table 2-1. The simulator consumes user and system traces while they are generated, and does not write them to non-volatile storage [10].

There are two main reasons for simulating the memory system from the DS5000/200. First, the DS5000/200 memory system is fairly conventional, with no unusual properties such as an extremely small TLB, virtually indexed cache, or untagged cache or TLB, that would reduce the generality of our results. Second, Mach 3.0 and Ultrix both run on the DS5000/200, allowing us to verify our simulation results against observed system behavior [11].

The two operating systems

Both operating systems implement the UNIX system call interface, although their underlying implementations differ substantially. Ultrix is a monolithic system in which all operating system code is implemented in the kernel. A program running on Ultrix invokes the operating system through a system call interface. In contrast, Mach 3.0 is a microkernel that exports and implements a small number of orthogonal abstractions including interprocess communication (IPC), threads, and virtual memory. Higher-level operating system services are implemented in a user-level process called the UNIX server. A program running on Mach 3.0 contacts the UNIX server through the Mach kernel's IPC interface [19], together with a user-level *transparent emulation library*, which is a shared library that is loaded into the address space of every process. The microkernel reflects UNIX system calls back to the calling program's emulation library, which converts the calls into RPCs to the UNIX server. (Simple UNIX services such as `getpid`() and signal masking are handled within the emulation library.) Virtual memory in Ultrix is derived

[1]The DECstation 5000/200 has a 25 MHz MIPS R3000 processor. Our machine used CPU version 2.0 implementation 2, and FPU version 2.0 implementation 3.

[2]The version numbers are: Ultrix V4.2 Rev. 96, Mach 3.0 MK78, CMU UNIX server UX39.

instruction cache: 64 KB, direct-mapped, physical, 16 byte line, 15 cycle miss penalty. **data cache:** 64 KB, direct-mapped, physical, 4 byte line, write allocate, 15 cycle read miss penalty, read miss fetches 16 aligned bytes. **write buffer:** six entries, page-mode writes complete in 1 cycle, non page-mode writes complete in 5 cycles; CPU reads have priority for memory access, but wait for writes that have already started. 4 KB page size for page-mode writes. **translation buffer:** 64 entries, 56 random/8 wired entries, trap to software on TLB miss. Each TLB entry maps a 4 KB page. **page mapping**: Deterministic. The physical page used to back a given virtual page is determined by the virtual page number and its address space identifier. The deterministic strategy prevents conflicts within any 64 KB (cache size) range of virtual addresses. **kernel memory**: All kernel text and most kernel data is in unmapped physical memory.

Table 2-1: Memory system simulation parameters.

from the original BSD abstractions [6], and is relatively machine-dependent. Mach uses a more flexible and aggressive virtual memory system which is partitioned into a machine-dependent and a machine-independent layer [33].

For our cross-system comparisons, the major UNIX components of Ultrix and CMU's server are similar but not identical. Although both systems are derived from the same code base, they have matured in different environments. We have nevertheless attempted to eliminate obvious superficial differences between the two systems. For example, both systems are compiled at the same optimization level with DEC's Ultrix compiler from MIPS Computer Systems,[3] and both systems use a large file buffer cache (12 MB).

2.1. Sources of distortion

A traced program is both larger (about a factor of two) and slower (about a factor of 15) than its untraced counterpart. The first effect, called *memory dilation*, can increase paging and TLB miss rates. We avoid perturbations due to paging by collecting our traces on a machine with a large physical memory so that pageouts do not occur. We contain TLB effects by simulating rather than tracing those TLB misses that could be affected by memory dilation. On the DECstation, there are two kinds of TLB misses. A TLB miss on a user virtual address (*UTLB* miss) is handled by a lightweight miss handler. This routine is not traced, but is simulated using references from the traces. A TLB miss to mapped kernel memory (*KTLB* miss) is handled by a more general TLB miss handler. The KTLB miss routine is traced, but the only KTLB misses that could be affected by tracing are those to page-table pages. Each page-table page, though, maps a four megabyte text segment, which is large enough to contain even our largest traced program.

Workload	Description	Mach time	Ultrix time
sed	The UNIX stream editor run three times over the same 17K input file.	0.58	0.57
egrep	The UNIX pattern search program run three times over a 27K input file.	2.01	1.90
yacc	The LR(1) parser-generator run on an 11K grammar.	1.75	1.82
gcc	The GNU C compiler (gcc) translating a 17K (preprocessed) source file into optimized Sun-3 assembly code.	3.70	4.20
compress	Data compression using Lempel-Ziv encoding. A 100K file is compressed then uncompressed.	1.32	1.32
ab	The Andrew Benchmark with *gcc*. The assembler was not traced.	112.18	98.96
espresso	A program that minimizes boolean function run on a 30K input file.	6.23	6.46
lisp	The 8-queens problem solved in LISP.	56.46	54.97
eqntott	A program that converts boolean equations to truth tables using a 1390 byte input file.	66.05	65.85
fpppp	A program that does quantum chemistry analysis. This program is written in Fortran.	25.20	16.78
doduc	Monte-Carlo simulation of the time evolution of a nuclear reactor component described by 8K input file. This program is written in Fortran.	22.94	24.56
liv	The Livermore Loops benchmark.	1.24	1.22
tomcatv	A program that generates a vectorized mesh. This program is written in Fortran.	139.42	155.44

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

Execution times (shown in seconds) in this table are for runs of an untraced binary on an untraced system. Except where indicated, all programs are written in C. The bottom four workloads are floating-point intensive. None of the programs have been reordered or tuned for the underlying memory system.

[3] The compiler is Version 2.1. The optimizations levels are those provided by the standard system makefiles (-O2), although some Ultrix device drivers are compiled without optimization.

The second effect, called *time dilation*, causes activity dependent on external events to appear to complete faster than in an uninstrumented system. To counter this, we have configured the system clock to interrupt at 1/15th the standard rate. We have not modified either system's I/O behavior to account for time dilation, as this would require subtle system changes that might themselves introduce other distortions. Instead, we separate the instruction reference stream into *non-idle* and *idle* references. Idle references, which occur during I/O operations as part of a system's idle loop in a uniprogrammed environment, are multiplied by the system code's expansion factor.

An address trace contains *virtual* addresses, yet the actual and simulated cache are indexed by *physical* addresses. A third source of distortion is therefore due to the simulator's model of the virtual memory system's page mapping strategy. The Ultrix mapping strategy, similar to the one described in Table 2-1, is deterministic. In contrast, Mach's strategy is random (a virtual page is bound to the first free physical page on the free list). This difference can have a measurable impact on system behavior (We isolate the effects of the page-mapping strategies in a later section.) Our simulator uses the deterministic mapping strategy described in Table 2-1, which improves the repeatability of simulation results, and eliminates a source of variability between the two systems.

2.2. Workloads and summary of results

A summary of the trace results for each program is shown in Table 2-3, with aggregate results shown in Table 2-4. For Ultrix, system behavior is confined to the kernel. For Mach, system behavior includes the kernel, the UNIX server, and the emulation library.

For a given workload, Ultrix issues more disk requests than Mach, resulting in greater idle instruction counts and delays. Ultrix is more conservative than Mach's UNIX server in forcing meta-data updates to disk. Additionally, programs under Mach are demand-paged, whereas under Ultrix they are loaded entirely at program startup, sometimes leading to unnecessary disk accesses. On the average, we saw about 1.4 times more I/O requests for workloads under Ultrix than for Mach. Because this difference in I/O behavior is orthogonal to the issue of kernel architecture, we exclude idle references from our remaining discussion.

For workloads that rely heavily on UNIX services, the combined Mach system components (microkernel, UNIX server, and emulation library) execute more instructions and generally require more data references than Ultrix.

Memory cycles per instruction

We use our simulation results to calculate *memory cycles per instruction* (*MCPI*), which is the number of CPU stall cycles due to the memory system divided by the number of instructions executed. *MCPI* is one of several components of cycles per instruction (*CPI*), which is a metric commonly used to evaluate computer systems [22]. Other components of *CPI*, such as one cycle per instruction for instruction execution, interlocks during multiply, divide, and floating point operations, and no-ops inserted by the compiler for load and branch delays, remain relatively constant even as processor cycle time decreases. In contrast, *MCPI* is a function of the ratio of memory speed to processor speed, is less dependent on processor architecture, and will dominate overall *CPI* if current trends in processor and memory speed continue. As mentioned, we have excluded idle-loop activity from our *MCPI* calculations. The idle loop rarely misses in the cache, so a system could achieve an artificially low *MCPI* by executing an arbitrarily large number of idle instructions.

The *MCPI* for each workload under Ultrix and Mach is shown in Figure 2-1. Each bar is shaded to denote different *MCPI* components, with the system and user contributions separated by a vertical bar. The figure shows that data and instruction cache misses in user and system mode are only partially responsible for the total *MCPI*. Other components include CPU write-stalls and kernel uncached memory reads. CPU write-stalls are reflected in Figure 2-1 in the *wbuffer* component, which shows the average per-instruction penalty from writes to a full write buffer, as well as reads that stall pending the completion of a five-cycle write. A system uncached memory read occurs when the kernel accesses memory through the uncached segment [24] such as for I/O or device control.

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

The counts from Table 2-3 along with the *MCPI*s from Figure 2-1 can be used to estimate actual run times for Ultrix, which uses a page mapping strategy similar to the simulator's. The cycle count, ignoring arithmetic interlocks, can be approximated by the *idle cycle count + non-idle cycle count*. Assuming an idle loop *CPI* of 1, the cycle count is then equal to the sum of the non-idle instruction count multiplied by (1 + *MCPI*) and the idle in-

	non-idle instructions				idle instructions		instruction cache misses				data cache reads				data cache read misses			
	Ultrix	%s	Mach	%s	Ultrix	Mach	Ultrix	%s	Mach	%s	Ultrix	%s	Mach	%s	Ultrix	%s	Mach	%s
sed	5704	24	7763	44	5876	1270	51	96	149	98	1515	17	2015	38	16	97	70	98
egrep	43277	4	45029	7	2495	914	43	93	140	98	8965	3	9425	8	32	91	71	97
yacc	32799	6	34539	10	13220	2809	69	89	166	96	6893	6	7283	11	48	50	93	71
gcc	29318	22	35939	36	63684	27027	485	42	999	71	8257	12	9941	27	120	44	318	71
compress	16896	19	19926	31	5555	2225	70	96	215	97	4778	12	5452	23	166	28	271	50
ab	869732	33	1198172	51	689324	247969	15612	52	28619	73	223588	26	295572	44	6658	79	11262	86
espresso	135385	2	137806	4	21601	8069	187	45	344	70	32034	2	32652	3	93	32	168	58
lisp	1288027	3	1276619	2	1005	0	222	61	2004	54	468287	2	467668	2	655	45	734	68
eqntott	1414369	1	1417868	1	10632	0	126	88	254	97	296691	1	297901	1	14328	3	14489	4
fpppp	265457	8	262998	7	17102	5667	4135	21	3735	19	139172	2	139036	2	131	27	177	47
doduc	321325	1	325351	2	18474	4983	6239	5	6292	7	122009	0	123020	1	550	9	612	21
liv	23008	3	23778	6	1585	639	21	93	72	98	7968	2	8176	4	17	88	30	96
tomcatv	2005703	1	2005590	1	10823	134	138	82	326	84	967474	0	968074	0	85451	0	85522	0

Table 2-3: Summary of trace results.

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

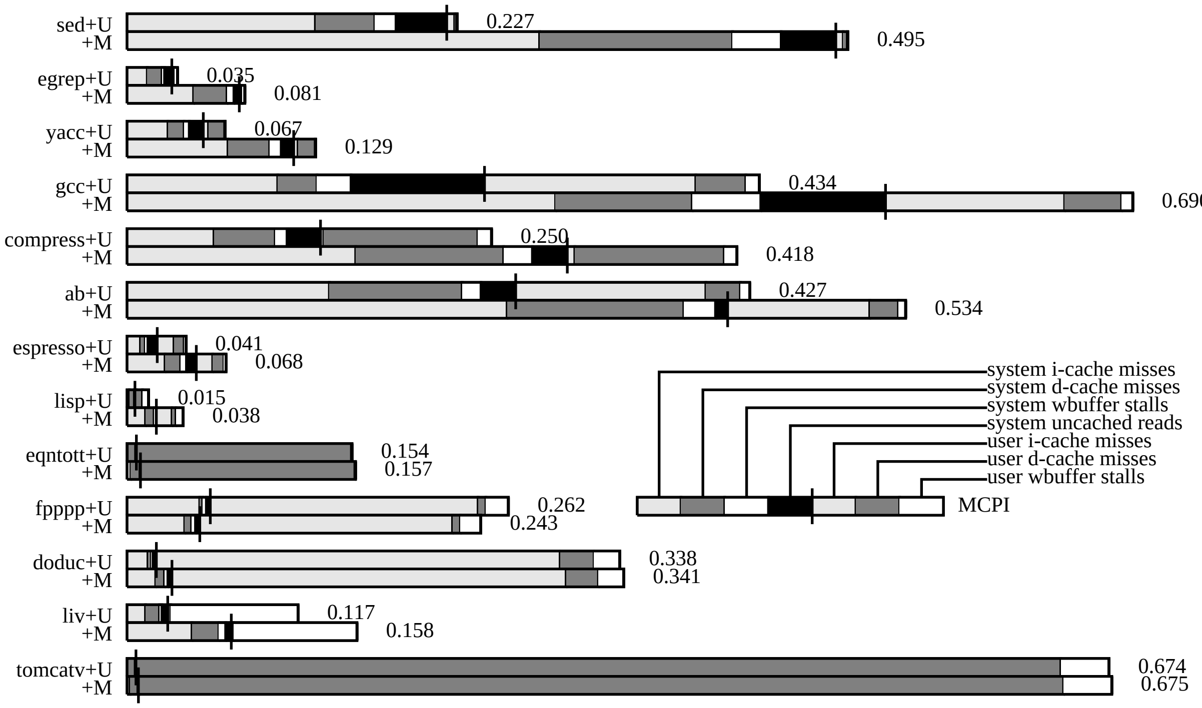

Figure 2-1: Baseline *MCPI* for Ultrix and Mach.

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

struction count. As an example, consider *gcc* with a cycle count of (29318000 × (1 + 0.434)) + (63684000) = 105726012. Dividing the cycle count by the clock speed (25 MHz), we compute a runtime of 4.22 seconds, which is close to the actual runtime (Table 2-2) of 4.20 seconds. Using *MCPI* to compute execution times for the Mach workloads is less accurate as Mach's page mapping strategy is non-deterministic.

3. Comparative system behavior

As shown in the previous section, the most significant difference between Mach and Ultrix is the number and cost of non-idle instructions required to run an application. In this section we discuss the influence that major system components have on system performance.

In Figure 3-1 we separate system overheads into 11 major components, and compare Ultrix and Mach in terms of these components. The components are: *trap* (system

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

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

These figures, which are the average over all the workloads, characterize system execution (non-idle system cycles / non-idle system instructions) vs. user execution (user cycles / user instructions). As such, they do not reflect the impact of system execution on overall performance.

call and exception handling), *UTLB* (user TLB miss), *KTLB* (kernel TLB miss), *VM-md* (machine-dependent virtual memory), *VM-mi* (Mach's machine independent virtual memory), *Block Ops* (block memory moves and zeroes), *UNIX service* (the remaining routines in the Ultrix kernel and Mach UNIX server), *Microkernel* (Mach's microkernel, including device management and scheduling), *IPC* (the Mach kernel's message system), *Emulator* (Mach's transparent emulation library), and *S-MCPI* (system memory cycles per system instruction).[4] The first ten categories reflect relative overheads in terms of non-idle system instructions executed. The last category, *S-MCPI*, reflects relative memory system overhead for system references. Four of the activities (*Microkernel, Emulator, IPC, VM-mi*) occur only in Mach. *Block Ops* for Mach includes operations from both the Mach kernel and the UNIX Server.

The number at the top of each bar is system cycles as a percentage of total cycles. The Ultrix instruction counts have been normalized to one for all workloads. The heights of the bars reflect *system*, but not *total* execution, overheads. For workloads where system activity is small relative to total activity (*doduc*, for example), system contribution to performance is minor.

Several characteristics of system behavior are worth noting from Figure 3-1. First, memory penalty for system instructions is from one to three times greater for Mach than for Ultrix. The difference in the system *MCPI*, while often small (Figure 2-1), can contribute substantially to overall system performance because of the large number of system instructions executed.

Second, Mach's virtual memory system executes more instructions than the one implemented in Ultrix, which has been flattened into a single machine-dependent layer. Mach has an additional machine-independent layer that is more costly than either systems' machine-dependent layer.

[4]*S-MCPI* is computed as the *system cycles / system instructions*, and differs from *MCPI* due to the system in that it includes only system instructions.

This comparison should not be taken to mean that one system is better or worse than the other, since Mach's virtual memory interface provides substantially more functionality and portability than Ultrix's.

A third point of comparison is the relative instruction cost of UNIX service in Ultrix, which is larger than that under Mach. For Ultrix, the *UNIX service* category includes many machine-dependent services such as device management that are counted as part of Mach's *Microkernel* category. On the other hand, the *Microkernel* category indicates that there is a measurable cost for providing those services through a separate set of kernel interfaces. The *UNIX service* category also includes some services that are implemented in Mach's transparent emulation library. For example, *lisp* has a relatively high *UNIX service* component under Ultrix, but almost none under Mach. This is because *lisp* frequently modifies UNIX signal state to support garbage collection, and signal state can be manipulated from within Mach's transparent emulation library.

Lastly, Figure 3-1 shows that the overhead of Mach's IPC, in terms of instructions executed, is responsible for a small portion of overall system overhead. This suggests that microkernel optimizations focusing exclusively on IPC [8, 18, 20, 26, 34], without considering other sources of system overhead such as *MCPI*, will have a limited impact on overall system performance [7].

4. Seven assertions

In this section we evaluate the strength of the seven assertions enumerated in the introduction. Our basic strategy is to address each assertion in the context of our traces. In several cases we present the results of additional simulations in which we vary the base architecture to determine the sensitivity of system performance to the assertion in question.

4.1. System and user locality

As cache behavior is an indication of locality, Table 2-3 supports the first assertion: *the operating system has less instruction and data locality than user programs.* The system can contribute up to 51% of non-idle instruction cache references, but in most cases (17 of 26) the system contribution is less than 10%. Given this, a disproportionately large number of instruction cache misses are due to the system (greater than 70% for two-thirds of the workload/system pairs).

In terms of data references, the system contributes a larger percentage of misses than references, again supporting the assertion that the system's data locality is worse

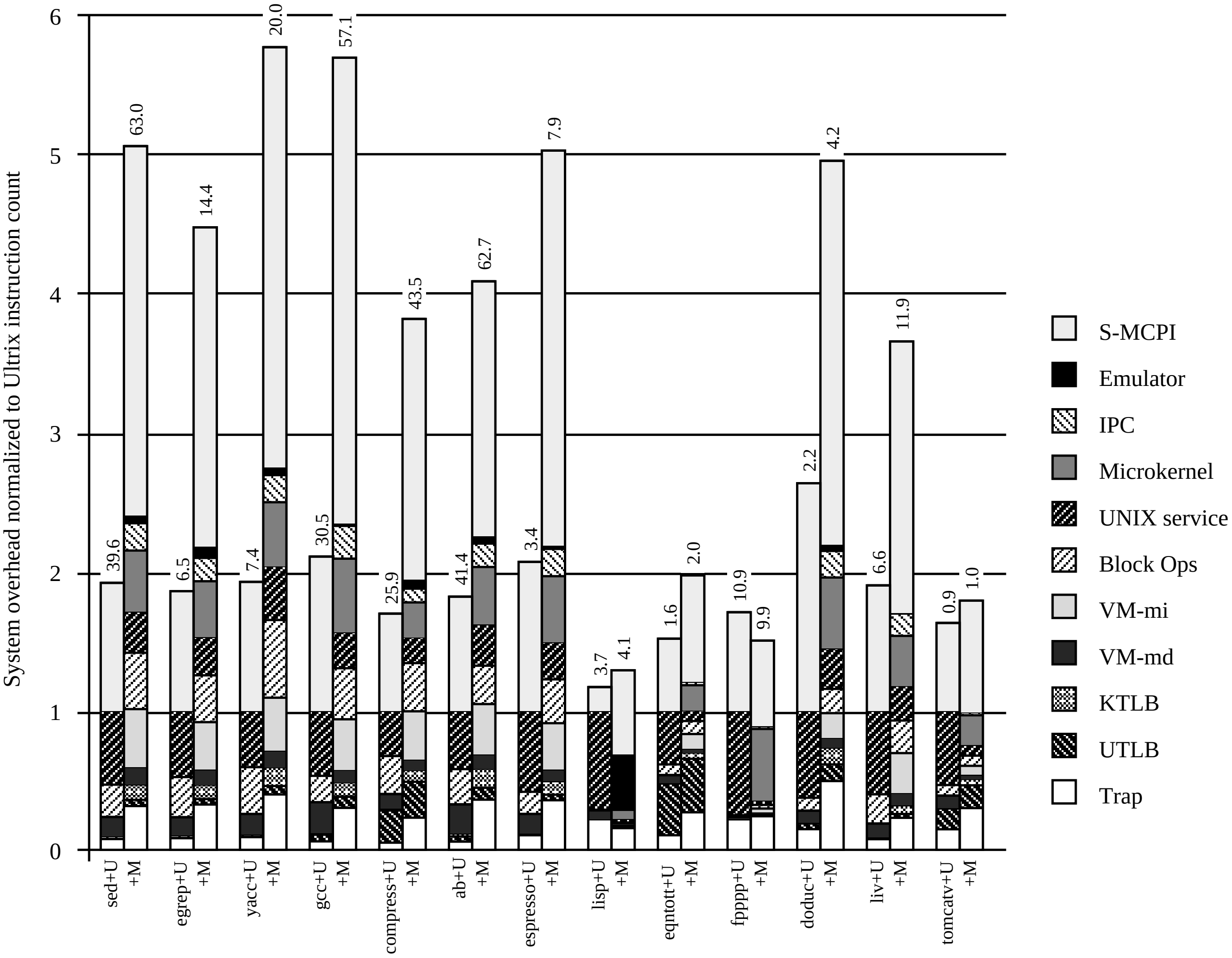

Figure 3-1: Relative system overheads for programs running on Ultrix and Mach.

This figure shows the relative system instruction and system memory overheads for programs running on Ultrix (+U) and Mach (+M). Ultrix instruction counts are normalized to one. The top component of each bar reflects the system *MCPI*, which is an aggregate of the *MCPI* for the instruction components. The number at the top of each bar is the percentage of total (instruction and memory) cycles that are due to the system. For programs where the system is responsible for a small percentage of total cycles, system overheads are relatively unimportant.

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

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

4.2. System instruction locality

Percentages are useful for comparing system and user behavior, but they cloud overall performance effects. For example, although 97% of instruction cache misses for *eqntott* under Mach are due to the system, the system instruction cache miss rate is insignificant.[5]

A better indicator of the performance impact of locality is the cache's contribution to *MCPI*. In Table 4-1 we combine our baseline data from Table 2-3 with cache miss penalties for the simulated memory system to yield the *MCPI* contributions from the cache. The component of *MCPI* due to system instruction cache references dominates that due to the user in 20 of 26 cases. In contrast, the system data cache component dominates in only thirteen cases. Furthermore, the majority of system (as opposed to user) cache penalties is due to poor instruction cache behavior; only five runs show greater penalties for data than instructions, and in these runs the system's contribution to *MCPI* is small. This behavior supports the second assertion: *system execution is more dependent on instruction cache behavior than is user execution.* However, many of the programs in our workload have small working sets that fit entirely in the instruction cache. Larger programs such as *gcc*, which do not fit in the cache, can have instruction cache penalties that rival that of the system.

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

Table 4-1: *MCPI* contributions from the cache.

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

Table 4-1 quantifies the difference in *MCPI* between Mach and Ultrix that was represented visually in Figure 2-1. Memory penalties due to system instruction and system data references are larger for Mach than for Ultrix, while user memory penalties are similar. Increased system activity in Mach, as is shown in Figure 3-1, results in a larger cache contribution to *MCPI*.

[5]The system instruction cache miss rates can be calculated with data from Table 2-3 as the *number of system instruction cache misses / number of system instruction cache references*. For example, for *eqntott*:

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

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

4.3. Competition between the user and system

The increased cache activity in Mach suggests that user code running on Mach may run more slowly than on Ultrix due to increased cache competition. To evaluate this, we ran the workloads against a simulated memory system that had independent 64 KB system and user caches. Again, by "system" for Mach, we mean the Mach kernel, the UNIX server, and the emulator.

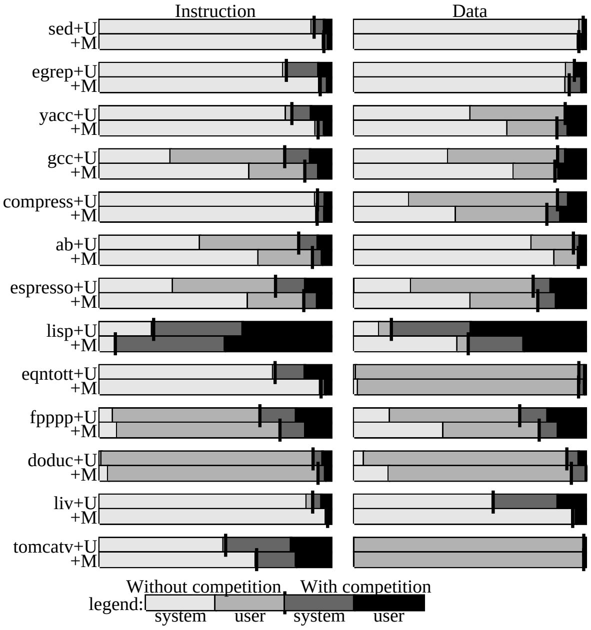

Figure 4-1: User/system interference.

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

The effects of user/system competition on instruction and data cache behavior for both systems are shown in Figure 4-1. Each instruction and data bar has four components. The two leftmost components correspond to the case of separate system and user caches, and represent the fraction of misses that remain with the independent caches. The two rightmost components show the additional fraction of system and user misses that occur when the cache is unified.

Although our separate user and system caches double the effective cache size, the general dominance of the two leftmost components in Figure 4-1 indicates that they do not significantly reduce miss rates relative to a smaller unified cache. The largest interference effects (for example, *lisp*) occur when the cache miss rate is low, such that a few interference misses can result in a large relative change. The absolute contribution of competition misses to *MCPI* is shown in Table 4-2. These points imply that the third assertion, *collisions between user and system references lead to significant performance degradation in the memory system*, is not true for these workloads.

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

Table 4-2: MCPI contributions from cache competition.

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

Voluntary context switches

In a client-server system such as Mach, voluntary context switches can occur every time the client and server interact through IPC. For the workloads we consider, the cache miss penalty following a voluntary context switch is not significant.[6] On the client side, where the instruction cache miss rates are generally low but data cache miss rates are high, the cost of reloading the cache after a context switch is amortized over a large number of instructions. On the server side, instruction and data locality are already poor, limiting the impact of interleaved user references. This behavior is consistent with earlier results on competition in client-server systems [28]. However, the penalty from competition clearly depends on the client-server system in question. Recent studies of the X11 window server, for example, have shown that larger programs and more frequent voluntary context switches create more severe penalties [12].

TLB behavior

The Ultrix kernel binary runs in unmapped kernel memory, largely isolating it from the TLB. In contrast, only Mach's microkernel component runs unmapped; the UNIX server and emulator run in mapped memory. Earlier research has shown that this structure can cause a significant increase in TLB activity [5, 30]. Table 4-3 confirms this, showing an order of magnitude increase in the number of system TLB references for Mach when compared to Ultrix.

In terms of *MCPI*, though, the absolute contribution of system TLB misses to performance is generally not large, shown by the last four columns of Table 4-3. Moreover, high TLB *MCPI* is an indication of poor locality, which is also reflected in more severe cache penalties. Even in runs with the most extreme behavior, TLB penalties are consistently dominated by cache penalties (Table 4-1) for both Ultrix and Mach.

4.4. System self-interference

Self-interference occurs when insufficient cache associativity results in cache misses. The impact of self-interference in user-code is well-understood [23]. To evaluate the impact of system self-interference, we simulated a two-way LRU set associative cache of the same size as our direct-mapped cache. As in the previous section, user references are isolated from the system-only cache, although they continue to generate TLB misses and subsequent system activity.

Figure 4-2 illustrates the effect of the increased system associativity on instruction and data miss rates. In each bar, the light region represents the fraction of system misses that associativity does not eliminate, while the dark region represents that fraction eliminated by associativity. This representation emphasizes variations in the relative benefit of associativity between workloads. The number at the left side of each bar is the absolute *MCPI* contribution of cache misses for a system-only direct-mapped cache. Figure 4-2 shows that the increased associativity eliminates a significant fraction of misses, and is more effective for instruction than data references. This confirms the fourth assertion: *self-interference is a problem in system instruction reference streams*.

Self-interference has the largest relative impact when *MCPI* is low, and the smallest relative impact when *MCPI* is high. A high *MCPI* implies that the cache is full, which is a situation that cannot be helped by increased associativity. For example, *sed*, *egrep*, and *liv* have high *MCPI*s, but gain relatively little from associativity. In contrast, associativity helps most with *lisp* and *tomcatv*, where *MCPI* is relatively low. Associativity is generally less beneficial for Mach than for Ultrix because applications on Mach tend to have a higher *MCPI*.

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

	TLB refs (x1000)			UTLB misses (x1000)				KTLB misses (x1)		UTLB MCPI		KTLB MCPI	
workload	user	Ultrix	Mach	U-user	M-user	Ultrix	Mach	Ultrix	Mach	Ultrix	Mach	Ultrix	Mach
sed	5596	423	1079	0.09	0.49	0.06	6.67	441	2132	0.000	0.012	0.021	0.063
egrep	50399	546	1116	0.07	0.39	0.10	6.41	472	1847	0.000	0.002	0.003	0.009
yacc	37460	571	1323	0.25	1.26	0.04	7.89	359	2280	0.000	0.003	0.003	0.015
gcc	30093	1582	2951	32.33	35.87	0.10	17.87	1521	3305	0.013	0.018	0.016	0.022
compress	17892	986	2085	82.58	82.06	0.12	10.24	712	3982	0.045	0.045	0.012	0.045
ab	755092	90958	195492	1148.79	1208.04	12.83	1457.98	95058	578598	0.014	0.025	0.030	0.108
espresso	164313	660	1281	0.86	2.64	0.05	7.67	452	3111	0.000	0.001	0.001	0.005
lisp	1706833	12974	26783	0.10	12.69	0.04	15.68	376	8063	0.000	0.000	0.000	0.002
eqntott	1690678	3579	3697	692.01	692.57	0.11	24.03	1321	9760	0.007	0.007	0.000	0.002
fpppp	380307	3632	1169	4.72	13.54	0.34	9.02	366	2273	0.000	0.001	0.000	0.003
doduc	438563	899	2162	16.78	30.53	0.04	18.26	402	5811	0.001	0.002	0.000	0.005
liv	30123	232	417	0.03	0.11	0.04	2.62	197	701	0.000	0.002	0.003	0.007
tomcatv	2949614	4480	2684	317.34	321.79	0.14	25.69	1608	8135	0.002	0.003	0.000	0.001

Table 4-3: TLB activity.

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

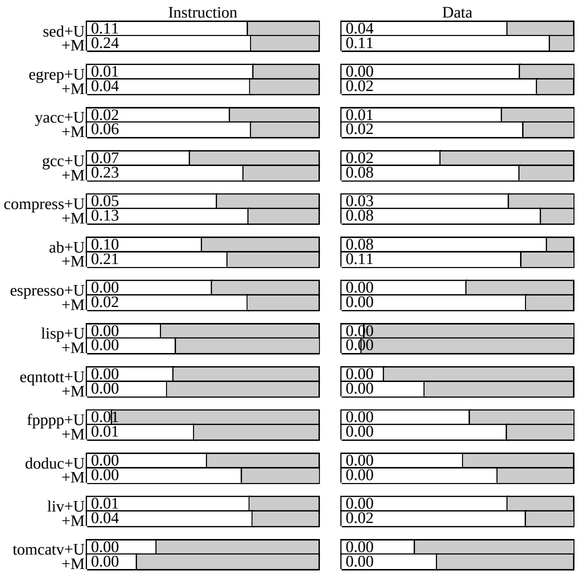

Figure 4-2: System self-interference.

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

4.5. Block operations

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

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

Table 4-4: MCPI from block memory operations.

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

In terms of *MCPI*, Table 4-4 shows that block operations incur a larger absolute overhead for programs running on Mach than on Ultrix. Table 4-5 shows that Mach generally references more data than Ultrix in block operations, and that more of those references go through to memory. Block operations in Mach occur within the kernel as part of the VM and IPC systems, and within the UNIX server as part of the file system. In contrast, Ultrix block operations, which occur entirely within the kernel, are due mostly to VM and file system operations.

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

Table 4-5: Block memory operations and memory reads.

For each system, this table shows the *MCPI* due to block memory operations and subsequent interference, and its percentage of total *MCPI* (Figure 2-1). The table also shows the number of data reads from cacheable and uncacheable memory that are due to block operations, the number of those reads that go to memory resulting in a CPU read stall, and the percentage of overall CPU memory stalls due to block operations. Reads from uncacheable memory are due primarily to I/O operations and always go through to memory. All counts are in thousands.

4.6. Streaming writes

Operating systems stream data to memory during block transfers, such as for I/O and IPC, and during context switches and exception handling. Write buffers expedite streaming writes by allowing the CPU to run ahead of memory. The effect of streaming write operations on system performance can be measured by counting stall cycles due to writes. The number of write stall cycles per instruction for user and system code under Ultrix and Mach is shown in Table 4-6. In most cases system behavior is worse than user behavior, supporting the sixth assertion: *write buffers are less effective for system references.*

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

Table 4-6: Write buffer stall cycles per instruction.

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

System write buffer stalls per instruction are generally higher for Mach than for Ultrix. Overall cache miss rates are higher with Mach, and the DECstation 5000/200 memory system gives CPU reads priority over outstanding writes. Consequently, fewer memory cycles are available for the write buffer to retire outstanding writes, resulting in a larger number of stalls. Additionally, the interleaved read misses decrease the frequency of low-latency page-mode writes.

4.7. Page mapping strategy

The system's virtual page mapping strategy can affect the performance of a physically indexed cache, as it determines the placement and overlap of virtual pages in the cache. As an example, the operating system can reduce self-interference misses for small applications by using a virtual-to-physical mapping that uniformly distributes consecutive virtual pages throughout the cache. For localities smaller than the cache size, such a strategy prevents collisions in the cache. This strategy also makes possible tools that rearrange the layout of text and data in memory to improve cache performance [27, 17].

In our discussion so far, we have simulated a deterministic strategy for both the Ultrix and Mach reference streams. As previously mentioned, Ultrix uses a deterministic strategy, while Mach's strategy is random (a virtual page is assigned to the next physical page on the free list). To isolate the effect of the page mapping strategy, we modified our simulator to use random mappings, and to maintain page tables so that page mappings do not change during a given run. Figure 4-3 shows *MCPI* for a run of the workloads with random page mapping. When compared to Figure 2-1 (both are on the same scale), most of the workload/system pairs perform better with random page mapping, and *gcc, compress, eqntott, fpppp, doduc* and *tomcatv* show the greatest improvement. The program *tomcatv* offers a good example of the effect that mapping strategy can have on program performance. This program uses several matrices that are rough multiples of the cache size, and allocated contiguously in virtual memory. The

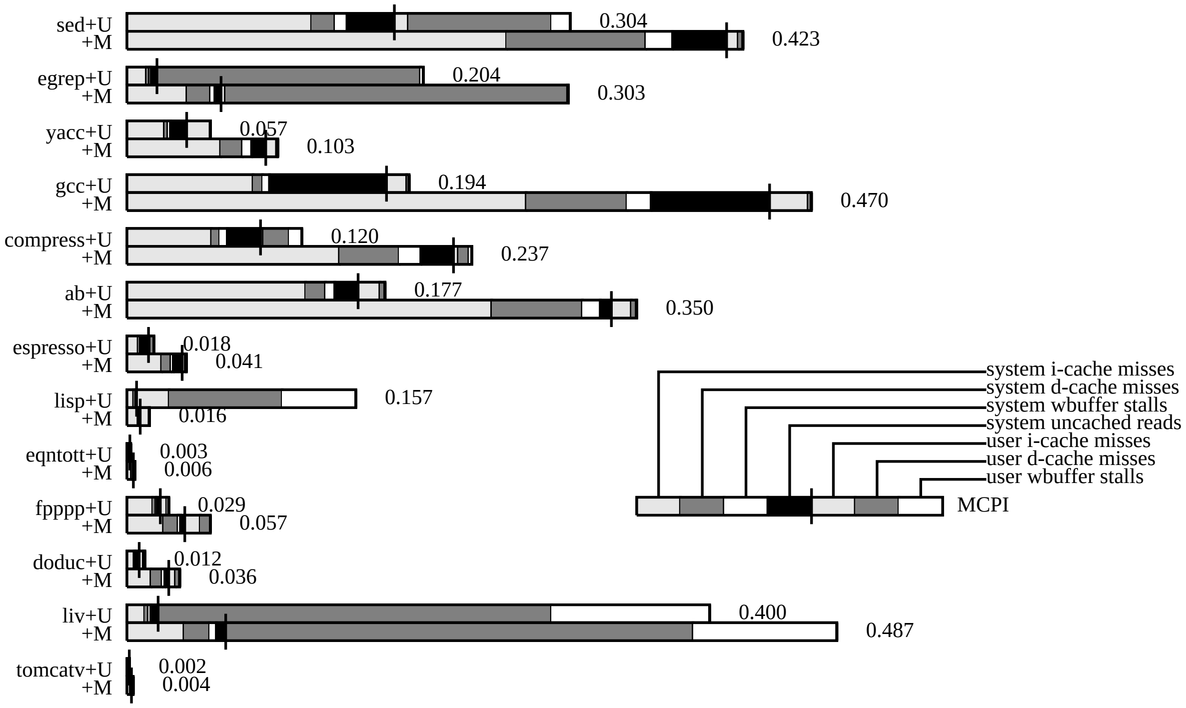

Figure 4-3: *MCPI* for random page mapping.

This figure shows *MCPI* for Mach and Ultrix, as in Figure 2-1, but for a system that uses random page mapping. The elimination of user cache misses reduces memory contention, so write buffer stalls are virtually eliminated for many workloads. This figure shows the results of a single run. As the page-mappings are random, behavior can vary significantly between runs.

virtual-to-physical mapping induced by the deterministic strategy causes frequent collisions between corresponding matrix elements during computation.

In some cases, the deterministic strategy yields a page mapping with low user cache miss rates. Specific examples are *sed* and *lisp* under Ultrix, and *egrep* and *liv* for both systems. In these cases the deterministic strategy leads to good behavior, and the random strategy can perform significantly worse. Our results, though, suggest that such cases are infrequent in the absence of program reordering.

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

5. Conclusions

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

- **System and user locality.** System locality is measurably worse than user locality, and the performance impact can be significant. The Mach microkernel-based system has poorer system locality than Ultrix.
- **System instruction locality.** Relative to user behavior, system text shows less locality than system data. However, user workloads such as *gcc* with large text can have instruction cache penalties that rival that of the operating system.
- **User/system competition.** User/system competition is a measurable component of cache and TLB miss rates. For these workloads, though, system performance is not affected by user/system competition. The impact of Mach's microkernel structure on competition is not significant.

- **System self-interference.** Self-interference accounts for a significant number of system misses, particularly in system text. However, the cases with the worst overall behavior are also those that benefit least from associativity. Compared to Ultrix, associativity eliminates a lower percentage of Mach's cache misses because of its greater demand for cache resources.
- **Block operations.** Block operations can be responsible for a large component of overall *MCPI*, particularly for applications that perform I/O. Mach moves more data with block operations and has a larger *MCPI* due to block operations than Ultrix.
- **Streaming writes.** System code presents a higher load to the write buffer than user code. Mach's increased cache *MCPI* results in a larger number of write buffer stalls due to competition between memory reads and writes.
- **Page mapping strategy.** Page mapping strategies can have a large effect on cache performance. The page mapping strategy is independent of operating system architecture.

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

6. Acknowledgements

Ali-Reza Adl-Tabatabai, Alan Eustace, Jay Lepreau, Kai Li, Stefan Savage, Daniel Stodolsky, Mark Swanson, Doug Tygar, and Terri Watson provided valuable feedback on earlier drafts of this paper. Alessandro Forin and Mary Thompson helped with our instrumentation of Mach. Bob Wheeler helped us understand the differences in I/O behavior between Mach and Ultrix. David Wall provided us with an initial version *epoxie*, and was helpful in making it work. The design of the tracing system is based on prior work by Anita Borg, and her contributions continued throughout this project.

References

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

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

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

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

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

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

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

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

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

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

11. J. Bradley Chen. Software Methods for System Address Tracing. The Proceedings of the Fourth Workshop on Workstation Operating Systems, October, 1993.

12. J. Bradley Chen. Memory Behavior for an X11 Window System. The Proceedings of the Winter 1994 USENIX Conference, January, 1994.

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

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

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

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

17. Digital Equipment Corporation. *cord*. Ultrix manual page.

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

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

20. Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto. The Proceedings of the 12th International Conference on Distributed Computing Systems, June, 1992.

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

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

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

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

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

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

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

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

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

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

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

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

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

34. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Giend, M. Guillemont, F. Herrmann, P. Leonard, S. Langlois, and W. Neuhauser. "Chorus Distributed Operating Systems". *Computing Systems 1*, 4 (1988), pp. 305-370.

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

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

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

Table of Contents

List of Figures

List of Tables