The Impact of Architectural Trends on Operating System Performance

Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod,
Emmett Witchel, and Anoop Gupta

Computer Systems Laboratory
Stanford University, Stanford CA 94305
http://www-flash.stanford.edu

Abstract

Computer systems are rapidly changing. Over the next few years, we will see wide-scale deployment of dynamically-scheduled processors that can issue multiple instructions every clock cycle, execute instructions out of order, and overlap computation and cache misses. We also expect clock-rates to increase, caches to grow, and multiprocessors to replace uniprocessors. Using SimOS, a complete machine simulation environment, this paper explores the impact of the above architectural trends on operating system performance. We present results based on the execution of large and realistic workloads (program development, transaction processing, and engineering compute-server) running on the IRIX 5.3 operating system from Silicon Graphics Inc.

Looking at uniprocessor trends, we find that disk I/O is the first-order bottleneck for workloads such as program development and transaction processing. Its importance continues to grow over time. Ignoring I/O, we find that the memory system is the key bottleneck, stalling the CPU for over 50% of the execution time. Surprisingly, however, our results show that this stall fraction is unlikely to increase on future machines due to increased cache sizes and new latency hiding techniques in processors. We also find that the benefits of these architectural trends spread broadly across a majority of the important services provided by the operating system. We find the situation to be much worse for multiprocessors. Most operating systems services consume 30-70% more time than their uniprocessor counterparts. A large fraction of the stalls are due to coherence misses caused by communication between processors. Because larger caches do not reduce coherence misses, the performance gap between uniprocessor and multiprocessor performance will increase unless operating system developers focus on kernel restructuring to reduce unnecessary communication. The paper presents a detailed decomposition of execution time (e.g., instruction execution time, memory stall time separately for instructions and data, synchronization time) for important kernel services in the three workloads.

1 Introduction

Users of modern computer systems expect the operating system to manage system resources and provide useful services with minimal overhead. In reality, however, modern operating systems are large and complex programs with memory and CPU requirements that dwarf many of the application programs that run on them. Consequently, complaints from users and application developers about operating system overheads have become commonplace.

The operating system developer's response to these complaints has been an attempt to tune the system to reduce the overheads. The key to this task is to identify the performance problems and to direct the tuning effort to correct them; a modern operating system is far too large to aggressively optimize each component, and misplaced optimizations can increase the complexity of the system without improving end-user performance. The optimization task is further complicated by the fact that the underlying hardware is constantly changing. As a result, optimizations that make sense on today's machines may be ineffective on tomorrow's machines.

In this paper we present a detailed characterization of a modern Unix operating system (Silicon Graphics IRIX 5.3), clearly identifying the areas that present key performance challenges. Our characterization has several unique aspects: (i) we present results based on the execution of large and realistic workloads (program development, transaction processing, and engineering compute-server), some with code and data segments larger than the operating system itself; (ii) we present results for multiple generations of computer systems, including machines that will likely become available two to three years from now; (iii) we present results for both uniprocessor and multiprocessor configurations, comparing their relative performance; and finally (iv) we present detailed performance data of specific operating system services (e.g. file I/0, process creation, page fault handling, etc.)

The technology used to gather these results is SimOS [11], a comprehensive machine and operating system simulation environment. SimOS simulates the hardware of modern uniprocessor and multiprocessor computer systems in enough detail to boot and run a commercial operating system. SimOS also contains features which enable non-intrusive yet highly detailed study of kernel execution. When running IRIX, SimOS supports application binaries that run on Silicon Graphics' machines. We exploit this capability to construct large, realistic workloads.

Focusing first on uniprocessor results, our data show that for both current and future systems the storage hierarchy (disk and memory system) is the key determinant of overall system performance. Given technology trends, we find that I/O is the first-order bottleneck for workloads such as program development and transaction processing. Consequently, any changes in the operating system which result in more efficient use of the I/O capacity would offer the most performance benefits.

After I/O, it is the memory system which has the most significant performance impact on the kernel. Contrary to expectations, we find that future memory systems will not be more of a bottleneck than they are today. Although memory speeds will not grow as rapidly as instruction-processing rates, the use of larger caches and dynamically-scheduled processors will compensate.

We find that on future machines, kernel performance will improve as fast as application program performance resulting in kernel overheads remaining relatively the same in the future. The important services of the kernel tend to benefit equally from improvements in execution speed so their relative importance remains unchanged in the future.

Looking at small-scale shared-memory multiprocessors, another likely architectural trend, we observe that the memory system behavior becomes even more important for overall perfor-

mance. We find that extra memory stall corresponding to communication between the processors (coherency cache misses) combined with synchronization overheads result in most multiprocessor operating system services consuming 30% to 70% more computational resources than their uniprocessor counterparts. Because larger caches do not reduce coherence misses, the performance gap between uniprocessor and multiprocessor performance will increase unless operating system developers focus on kernel restructuring to reduce unnecessary communication.

The rest of the paper is organized as follows. Section 2 presents our experimental environment, including SimOS, workloads, and data collection methodologies. Section 3 describes the current and future machine models used in this study. Sections 4 and 5 present the experimental results for the uniprocessor and multiprocessor models. Finally, Section 6 discusses related work and Section 7 concludes.

2 Experimental Environment

In this section, we present the SimOS environment, describe our data collection methodology, and present the workloads used throughout this study.

2.1 The SimOS Simulation Environment

SimOS [11] is a machine simulation environment that simulates the hardware of uniprocessor and multiprocessor computer systems in enough detail to boot, run, and study a commercial operating system. Specifically, SimOS provides simulators of CPUs, caches, memory systems, and a number of different I/O devices including SCSI disks, ethernet interfaces, and a console.

The version of SimOS used in this study models the hardware of machines from Silicon Graphics. As a result, we use Silicon Graphics' IRIX 5.3 operating system, an enhanced version of SVR4 Unix. This version of IRIX has been the subject of much performance tuning on uniprocessors and on multiprocessors with as many as 36 processors. Although the exact characterization that we provide is specific to IRIX 5.3, we believe that many of our observations are applicable to other well-tuned operating systems.

Although many machine simulation environments have been built and used to run complex workloads, there are a number of unique features in SimOS that make detailed workload and kernel studies possible:

Multiple CPU simulators. In addition to configurable cache and memory system parameters typically found in simulation environments, SimOS supports a range of compatible CPU simulators. Each simulator has its own speed-detail trade-off. For this study, we use an extremely fast binary-to-binary translation simulator for booting the operating system, warming up the file caches, and positioning a workload for detailed study. This fast mode is capable of executing workloads less than 10 times slower than the underlying host machine. The study presented in this paper uses two more detailed CPU simulators that are orders of magnitude slower than the fastest one. Without the fastest simulator, positioning the workloads would have taken an inordinate amount of time. For example, booting and configuring the commercial database system took several tens of billion of instructions which would have taken several months of simulation time on the slowest CPU simulator.

Checkpoints. SimOS can save the entire state of its simulated hardware at any time during a simulation. This saved state, which includes the contents of all registers, main memory, and I/O devices, can then be restored at a later time. A single checkpoint can be restored to several different machine configurations, allowing the workload to be examined running on different cache and CPU parameters. Checkpoints allow us to start each workload at the point of interest without wasting time rebooting the operating system and positioning the applications.

Annotations. To better observe workload execution, SimOS supports a mechanism called *annotations* in which a user-specified routine is invoked whenever a particular event occurs. Most annotations are set like debugger breakpoints so they trigger when the workload execution reaches a specified program counter address. Annotations are non-intrusive. They do not effect workload execution or timing, but have access to the entire hardware state of the simulated machine.

2.1.1 Data Collection

Because SimOS simulates all the hardware of the system, a variety of hardware-related statistics can be kept accurately and non-intrusively. These statistics cover instruction execution, cache misses, memory stall, interrupts, and exceptions. The simulator is also aware of the current execution mode of the processors and the current program counter. However, this does not provide information on important aspects of the operating system such as the current process id or the service currently being executed.

To further track operating system execution, we implement a set of state machines (one per processor and one per process) and one pushdown automata per processor to keep track of interrupts. These automata are driven by a total of 67 annotations. For example, annotations set at the beginning and end of the kernel idle loop separate idle time from kernel execution time. Annotations in the context switch, process creation, and process exit code keep track of the current running process. Since they have access to all registers and memory of the machine, they can non-intrusively determine the current running process id and its name. Additional annotations are set in the page fault routines, interrupt handlers, disk driver, and at all hardware exceptions. These are used to attribute kernel execution time to the service performed. Annotations at the entry and exit points of the routines that acquire and release spin locks determine the synchronization time for the system, and for each individual spin lock.

Additionally, we maintain a state machine per line of memory to track cache misses. These state machines allows us to report the types of cache misses (i.e. cold, capacity, invalidation, etc.) and whether the miss was due to interference between the kernel and user applications. We also track cache misses and stall time by the program counter generating the misses and by the virtual address of the misses. This allows us to categorize memory stall both by the routine and the data structure that caused it.

2.1.2 Simulator Validation

One concern that needs to be addressed by any simulation-based study is the validity of the simulator. For an environment such as SimOS, we must address two potential sources of error. First, we must ensure that when moving the workloads into the simulation environment we do not change their execution behavior. Additionally, we must ensure that the timings and reported statistics are correct. Establishing that SimOS correctly executes the workload is fairly straightforward.

First, the code running on the real machine and SimOS are basically identical. The few differences between the IRIX kernel and its SimOS port are mostly due to the I/O device drivers that communicate with SimOS' timer chip, SCSI bus, and ethernet interface. This code is not performance critical and tends to be different on each generation of computer anyway. All user-level code is unmodified.

Because SimOS simulates the entire machine, it's difficult to imagine these complex workloads completing correctly without performing the same execution as on the real machine. As further validation of correct execution, we compare workloads running on a Silicon Graphics POWER Series multiprocessor and a similarly

configured SimOS. At the level of the system call and other traps recorded by IRIX, the counts were nearly identical, and the differences are easily accounted for.

A second potential source of error is in the environment's timing and the statistics collection. This kind of error is more difficult to detect since it is likely the workload will continue to run correctly. To validate the timings and statistic reporting, we configure SimOS to look like the one-cluster DASH multiprocessor used in a previous operating system characterization study [2] and examine the cache and profile statistics of a parallel compilation workload. Statistics in [2] were obtained with a bus monitor, and are presented in Table 2.1. Although the sources of these statistics are completely different, the system behavior is quite similar.

	Execution profile			Fraction of misses in kernel mode
	Kernel	User	Idle	
SimOS	25%	53%	22%	52%
Bus monitor [2]	24%	48%	28%	49%

TABLE 2.1. SimOS validation results.
We compare several coarse statistics from SimOS to a published operating system characterization. Workload profiles match quite closely, and we attribute the reduced idle time in SimOS to a slightly more aggressive disk subsystem.

The absence of existing systems with dynamically-scheduled processors makes validation of the next-generation machine model difficult. However, the workloads do execute correctly, producing the same results as the single-issue CPU model. While these validation exercises are not exhaustive, they provide confidence in the simulation environment by showing that SimOS produces results comparable to earlier experiments

2.2 Workloads

Workload selection plays a large part in exposing operating system behavior. Our choice of workloads reflects a desire to investigate realistic applications found in a variety of computing environments. The three workloads that we use represent program development, commercial data processing, and engineering environments. Each workload has a uniprocessor and an eight-CPU multiprocessor configuration.

For each workload, we first boot the operating system and then log onto the simulated machine. Because operating systems frequently have significant internal state that accumulates over time, running the workloads directly after booting would expose numerous transient effects that do not occur in operating systems under standard conditions. To avoid these transient effects, we ensure in our experiments that kernel-resident data structures, such as the file cache and file system name translation cache, are warmed up and in a state typical of normal operation. We accomplish this either by running the entire workload once, and then taking our measurements on the second run, or by starting our measurements once the workload had run long enough to initialize the kernel data structures on its own.

Program Development Workload. A common use of today's machines is as a platform for program development. This type of workload typically includes many small, short-lived processes that rely significantly on operating system services. We use a variant of the compile phase of the Modified Andrew Benchmark [10]. The Modified Andrew Benchmark uses the gcc compiler to compile 17 files with an average length of 427 lines each. Our variant reduces the final serial portion of the make to a single invocation of the archival maintainer (we removed another invocation of *ar* as well as the cleanup phase where object files are deleted).

For the uniprocessor case, we use a parallel make utility configured to allow at most two compilation processes to run at any given time. For the eight-CPU multiprocessor case, we launch four parallel makes, and each allows up to four concurrent compilations. Each make performs the same task as the uniprocessor version, and on the average, we still maintain two processes per processor. To reduce the I/O bottleneck on the /tmp directory, we assign separate temporary directories (each on a separate disk device) to each make.

Database Workload. As our second workload, we examine the performance impact of a Sybase SQL Server (version 10 for SGI IRIX) supporting a transaction processing workload. This workload is a bank/customer transaction suite modeled after the TPC-B transaction processing benchmark [4]. The database consists of 63 Mbytes of data and 570 Kbytes of indexes. The data and the transaction logs are stored on separate disk devices. This workload makes heavy use of the operating system, specifically inter-processor communication.

In the uniprocessor version of this workload, we launch 20 client processes that request a total of 1000 transactions from a single server. For the multiprocessor workload, we increase the number of server engines to 6 and drive these with 60 clients requesting a total of 1000 transactions. The database log is kept on a separate disk from the database itself. The multiprocessor database is striped across 4 disks to improve throughput.

Engineering Workload. The final workload we use represents an engineering development environment. Our workload combines instances of a large memory system simulation (we simulate the memory system of the Stanford FLASH machine [7] using the FlashLite simulator) along with verilog simulation runs (we simulate the verilog of the FLASH MAGIC chip using the Chronologics VCS simulator). These applications are not operating system intensive because they do few system calls and require few disk accesses, but their large text segments and working sets stress the virtual memory system of the machine. This workload is extremely stable, and so we examine just over four seconds of execution.

The uniprocessor version runs one copy of FlashLite and one copy of the VCS simulator. The multiprocessor version runs six copies of each simulator.

3 Architectural Models

One of the primary advantages of running an operating system on top of a machine simulator is that it is possible to examine the effects of hardware changes. In this paper we use the capabilities of SimOS to model several different hardware platforms. This section describes three different configurations which correspond to processor chips that first shipped in 1994, and chips that are likely to ship in 1996 and 1998. Additionally, we describe the parameters used in our multiprocessor investigations.

3.1 Common Machine Parameters.

While we vary several machine parameters, there are others that remain constant. All simulated machines contain 128 Mbytes of main memory, support multiple disks, and have a single console device. The timing of the disk device is modeled using a validated simulator of the HP 97560 disk[1] [6]. Data from the disk is transferred to memory using cache-coherent DMA. No input is given to the console and the ethernet controller during the measurement runs. The CPU models support the MIPS-2

1. We found that the performance of the database workload was completely I/O bound using the standard disk model incorporated into SimOS. Given that these disks do not represent the latest technology, we scale them to be four times faster in the database workload.

	Machine Model		
	1994	**1996**	**1998**
CPU Clock	200Mhz	200Mhz	500Mhz
Pipeline	MIPS R4400-like Statically-scheduled Blocking caches	MIPS R10000-like Dynamically-scheduled Non-blocking caches	
Peak Performance	200 MIPS	800 MIPS	2000 MIPS
L1 Cache (Instructions)	16 KB, 2-way, 16 byte lines	32 KB, 2-way, 64 byte lines	64 KB, 2-way, 64 byte line
L1 Cache (Data)	16 KB, 2-way, 16 byte lines	32 KB, 2-way, 32 byte lines	64 KB, 2-way, 32 byte lines
L2 Cache (Unified)	1 MB, 1-way 128 byte lines	1 MB, 2-way, 128 byte lines	4 MB, 2-way, 128 byte lines
L1 miss/ L2 hit time	50 nanosecs	50 nanosecs	30 nanosecs
L2 miss time	500 nanosecs	300 nanosecs	250 nanosec

TABLE 3.1. 1994, 1996, and 1998 machine model parameters.
The peak performance is achieved in the absence of memory or pipeline stalls. The timings are the latency of the miss as observed by the processor. All 2-way set associative caches use an LRU replacement policy.

instruction set. The memory management and trap architecture of the CPU models are that of the MIPS R3000. Memory management is handled by a software-reload TLB configured with 64 fully-associative entries and a 4 kilobyte page size.

3.2 1994 Model

We base the 1994 model on the Indigo line of workstations from Silicon Graphics which contain the MIPS R4400 processor. The R4400 uses a fairly simple pipeline model that is capable of executing most instructions in a single clock cycle. It has a two level cache hierarchy with separate level-1 instruction and data caches on chip, and an off-chip unified level-2 cache. The MIPS R4400 has *blocking* caches. When a cache miss occurs the processor stalls until the miss is satisfied by the second level cache or memory system.

To model the R4400, we use a simple simulator which executes all instructions in a single cycle. Cache misses in this simulator stall the CPU for the duration of the cache miss. Cache size, organization, and miss penalties were chosen based on the SGI workstation parameters.[1]

3.3 1996 Model

Next-generation microprocessors such as the MIPS R10000 [9], Intel P6, and Sun UltraSPARC, will incorporate several new features including multiple instruction issue, dynamic scheduling, and non-blocking caches. The *multiple instruction issue* feature allows these processors to issue multiple consecutive instructions every clock cycle. *Dynamic scheduling* allows the instructions within a certain window to be shuffled around and issued out of order to the execution units, as long as essential dependences are maintained. This technique allows greater concurrency to be exploited in executing the instruction stream. With *branch prediction*, it is also possible to speculatively execute past branches whose outcome is yet unknown. Finally, *non-blocking caches* allow multiple loads and stores that miss in the cache to be serviced by the memory system simultaneously. Non-blocking caches, coupled with dynamic scheduling, allow the execution of any available instructions while cache misses are satisfied. This ability to hide cache miss latency is potentially a large performance win for programs with poor memory system locality, a characteristic frequently attributed to operating system kernels.

We model these next-generation processors using the MXS CPU simulator [1]. We configure the MXS pipeline and caches to model the MIPS R10000, the successor to the MIPS R4400 due out in early 1996.

The MXS simulator models a processor built out of decoupled fetch, execution, and graduation units. The fetch unit retrieves up to 4 instructions per cycle from the instruction cache into a buffer called the instruction window. To avoid waiting for conditional branches to be executed, the fetch unit implements a branch prediction algorithm that allows it to fetch through up to 4 unresolved conditional branches and register indirect jumps.

As the fetch unit is filling the instruction window, the execution unit is scanning it looking for instructions that are ready to execute. The execution unit can begin the execution of up to 4 instructions per cycle. Once the instruction execution has complete, the graduation unit removes the finished instruction from the instruction window and makes the instruction's changes permanent (i.e. they are committed to the register file or to the cache). The graduation unit graduates up to 4 instructions per cycle. To support precise exceptions, instructions are always graduated in the order in which they were fetched.

Both the level-1 and level-2 caches are non-blocking and support up to four outstanding misses. The level-1 caches support up to two cache accesses per cycle even with misses outstanding.

With cache miss stalls being overlapped with instruction execution and other stalls, it is difficult to precisely define a memory stall. When the graduation unit cannot graduate its full load of four instructions, we record the wasted cycles as stall time. We further decompose this stall time based on the state of the graduation unit. If the graduation unit cannot proceed due to a load or store instruction that missed in the data cache, we record this as data cache stall. If the entire instruction window is empty and the fetch unit is stalled on an instruction cache miss, we record an instruction cache stall. Finally, any other condition is attributed to pipeline stall because it is normally caused by pipeline dependencies.

Although MXS models the latencies of the R10000 instructions, it has some performance advantages over the real R10000. Its internal queues and tables are slightly more aggressive than the R10000. The reorder buffer can hold 64 instructions, the load/store queue can hold 32 instructions, and the branch prediction table has 1024 entries. Furthermore, it does not contain any of the execution-unit restrictions that are present in most of the next-generation processors. For example, the R10000 has only one shifter functional unit, so it can execute only one shift instruction per cycle. MXS can execute any four instruction per cycle including four shift instructions. We use this slightly more aggressive model in order to avoid the specifics of the R10000 implementation and provide results that are more generally applicable. Additional parameters of the 1996 model are presented in Table 3.1.

3.4 1998 Model

It is difficult to predict the architecture and speeds of the processors that will appear in 1998 since they haven't been announced yet. Processors like the MIPS R10000 have significantly increased the complexity of the design while holding the clock rate relatively constant. The next challenge appears to be increasing the clock rate without sacrificing advanced processor features [5]. We assume that a 1998 microprocessor will contain the latency tolerating features of the 1996 model, but will run at a 500Mhz clock rate and contain larger caches. We also allow for

1. The R4400 level-1 caches are direct mapped, but the newer R4600 has two-way set associative level-1 caches. We conservatively choose to model two-way set associativity in our level-1 caches.

small improvements in cache and memory system miss times. The exact machine parameters are again shown in Table 3.1.

3.5 Multiprocessor Model

Another trend in computer systems is to have multiple CPUs sharing a common memory. Although shared-memory multiprocessors have been around for a long time, recent microprocessor trends have the potential of making these systems much more common. Many next generation microprocessors, such as the MIPS R10000 and the Intel P6, support “glue-less MP” where shared memory multiprocessors can be built simply by plugging multiple CPUs into a shared bus.

Our multiprocessor studies are based on an 8-CPU system with a uniform memory access time shared memory. We use 1994 model processors; multiprocessor studies with the MXS simulator were prohibitively time consuming. Each CPU has its own caches. Cache-coherency is maintained by a 3-state (invalid, shared, dirty) invalidation-based protocol. The cache access times and main memory-latency are modelled to be the same as those in the 1994 model.

4 Uniprocessor Results

The vast majority of machines on the market today are uniprocessors, and this is where we start our examination of operating system performance. In this section we present a detailed characterization of the workloads running on the three machine configurations. In Section 4.1, we begin by describing the performance on the 1994 machine model. We then show in Section 4.2 how the 1996 and 1998 models improve performance. In Section 4.3 and Section 4.4, we show the specific impact of two architectural trends: latency hiding mechanisms and increases in cache size. Finally, in Section 4.5 we present a detailed examination of the relative call frequency, computation time, memory system behavior, and scaling of specific kernel services.

4.1 Base Characterization

Table 4.2 describes the operating system and hardware event rates for the workloads. In Figure 4.1, we provide a time-based profile of the execution of the workloads

The program development workload makes heavy but erratic use of the kernel services resulting in 16% of the non-idle execution time being spent in the kernel. The frequent creation and deletion of processes result in the large spikes of kernel activity found in the profile. The workload also generates a steady stream of disk I/Os, but contains enough concurrency to overlap most of the disk waits. As a result, the workload shows only a small amount of idle time.

The database workload makes heavy use of a number of kernel services. Inter-process communication occurs between the clients and the database server and between the database server and its asynchronous I/O processes. The result of this communication is both a high system call and context-switching rate. These effects, combined with a high TLB miss rate, result in the kernel occupying 38% of the non-idle execution time. The database workload also makes heavy use of the disks. Data is constantly read from the database’s data disk and log entries are written to a separate disk. Although the server is very good at overlapping computation with the disk operations, the workload is nevertheless idle for 36% of the execution time.

The engineering workload uses very few system services. Only the process scheduler and TLB miss handler are heavily used, and the kernel accounts for just 5% of the total workload execution time. The comb-like profile is due to the workload switching between the VCS and Flashlite processes, each of which has very different memory system behavior.

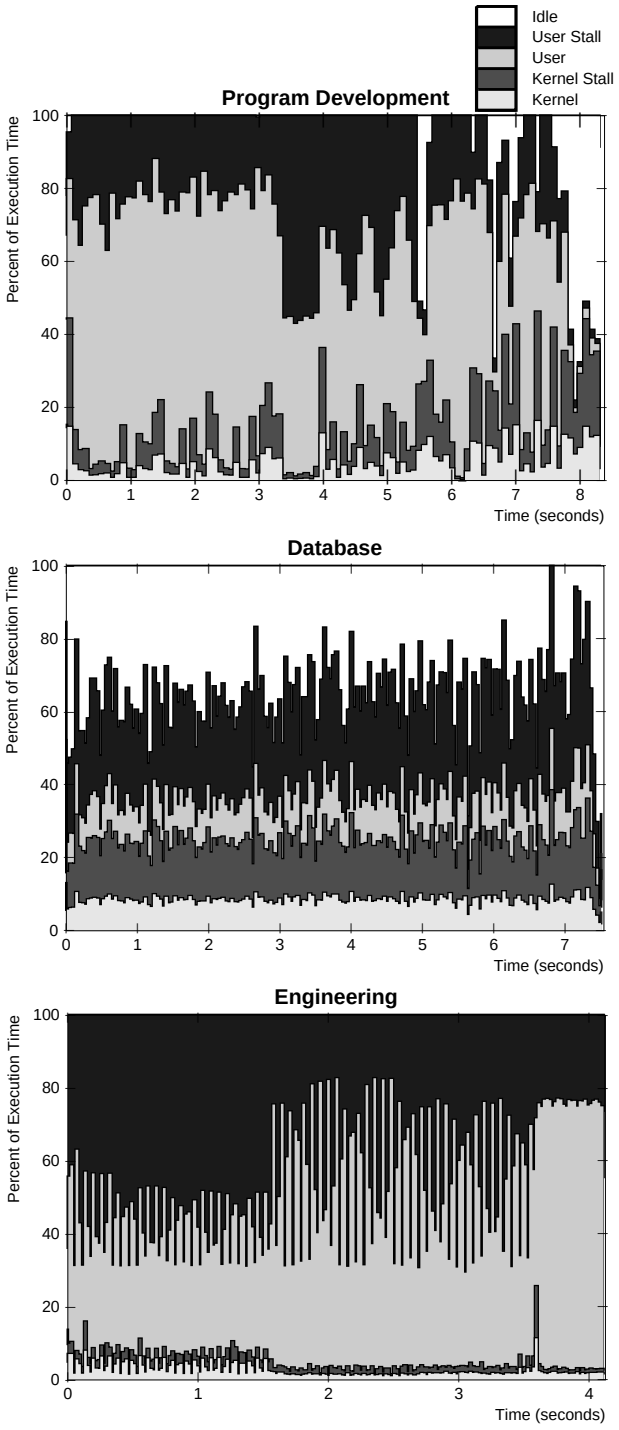

FIGURE 4.1. Profiles of uniprocessor workloads.
The execution time of each workload is separated into the time spent in user, kernel, and idle modes on the 1994 model. User and kernel modes are further subdivided into instruction execution and memory stall.

Also visible in Figure 4.1 is the large amount of memory stall time present in all of the workloads. Memory stall time is particularly prevalent in the database and engineering workloads, the two workloads that consist of large applications.

OS events	Prog-Dev	Database	Eng
Duration	8.5 secs	7.6 secs	4.1 secs
Process creations	11	< 1	< 1
Context switches	92	847	34
Interrupts	162	753	133
System calls	1133	4632	18
TLB refills	87 x 10^3	425 x 10^3	486 x 10^3
VM faults	2195	9197	3386
Other exceptions	405	304	12
Hardware events			
Instructions	129 x 10^6	111 x 10^6	101 x 10^6
L1-I cache misses	2738 x 10^3	4441 x 10^3	4162 x 10^3
L1-D cache misses	1412 x 10^3	1453 x 10^3	1628 x 10^3
L2-cache misses	324 x 10^3	339 x 10^3	460 x 10^3
Disk I/Os	29	286	1

TABLE 4.2. Event rates for the uniprocessor workloads.
All rates are reported as events per second on the 1994 model.

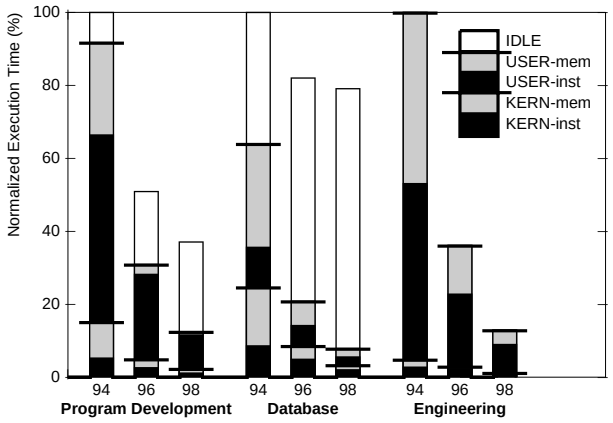

FIGURE 4.3. Execution time on next-generation machines.
This figure shows execution time of the three workloads running on the three machine models. The time is normalized to the speed of the 1994 model. The horizontal bars separate kernel, user, and idle time. Note that the engineering workload has no idle time and little kernel time.

4.2 Impact of Next-Generation Processors

In this section we examine the effect of future architectures on the three workloads. Figure 4.3 shows the normalized execution time of the workloads as the machine model is changed from the 1994 to the 1996 and 1998 models. The speedups for the 1998 model range from a fairly impressive factor of 8 for the engineering workload to a modest 27% for the database.

The primary cause of the poor speedup is delays introduced by disk accesses. This is the classic I/O bottleneck problem and can be seen in the large increases in idle time for the workloads with significant disk I/O rates. For the database system, the fraction of the execution time spent in the idle loop increases from 36% of the workload on the 1994 model to 75% of the time in the 1996 model and over 90% of the 1998 model. The program development workload also suffers from this problem with the 1998 model spending over 66% of the time waiting in the idle loop for disk requests to complete.

The implications of this I/O bottleneck on the operating system are different for the database and program development workloads. In the database workload, almost all of the disk accesses are made by the database server using the Unix "raw" disk device interface. This interface bypasses the file system allowing the data server to directly launch disk read and write requests. Given this usage, there is little that the operating system can do to reduce the I/O time. Possible solutions include striping the data across multiple disks or switching to RAIDs and other higher performance disk subsystems.

In contrast, the kernel is directly responsible for the I/O-incurred idle time present in the program development workload. Like many other Unix file systems, the IRIX extent-based file system uses synchronous writes to update file system meta-data structures whenever files are created or deleted. The frequent creation and deletion of compiler temporary files results in most of the disk traffic being writes to the meta-data associated with the temporary file directory. Almost half of the workload's disk requests are writes to the single disk sector containing the `/tmp` meta-data! There have been a number of proposed and implemented solutions to the meta-data update problems. These solutions range from special-casing the `/tmp` directory and making it a memory-based file system to adding write-ahead logging to file

Ignoring idle time[2], the computation portions of the workloads all show significant speedups. The advanced features of the 1996 model give it non-idle time speedups of 2.8x (Engineering), 3.0x (Program Development), and 3.1x (Database). The larger caches and higher clock rate of the 1998 model result in non-idle speedups of 7.4x (Program Development), 7.9x (Engineering), and 8.3x (Database).

Figure 4.3 highlights two other important points. First, the overall performance gains of the future machine models appear to apply equally to both user and kernel code. This implies that the relative importance of kernel execution time will likely remain the same on next-generation machines. While this means that the kernel time will remain significant, it is certainly preferable to increased kernel overhead.

Second, the fraction of execution time spent in memory stalls does not increase on the significantly faster CPUs. This is a surprising result given the increase in peak processor performance. Figure 4.4 shows the memory stall time on future machines expressed as memory stall cycles per instruction(MCPI). We see that next-generation machines have a significantly smaller amount of memory stall time than the 1994 model. This is indeed fortunate since the 1996 and 1998 models can execute up to 4 instruction per cycle, making them much more sensitive to large stall times. If the 1996 and 1998 models had the 1994 model's MCPI, they would spend 80% to 90% of their time stalled.

Figure 4.4 also decomposes the MCPI into instruction and data cache stalls and into level-1 and level-2 cache stalls. Although instruction cache stalls account for a large portion of the kernel stall time on the 1994 model, the effect is less prominent on the 1996 and 1998 models. For the program development workload, the instruction cache stall time is reduced from 45% of the kernel stall time in the 1994 model to only 11% of the kernel stall time in the 1998 model.

Figure 4.4 emphasizes the different memory system behavior of the workloads. The relatively small processes that comprise the program development workload easily fit into the caches of future

1. SGI's new file system, XFS, contains write-ahead logging of meta-data. Unfortunately, XFS was not available for this performance study.
2. To ensure that this omission does not compromise accuracy, we examine the program development and database workloads with disks that were 100 times faster. We found little differences in the non-idle memory system behavior.

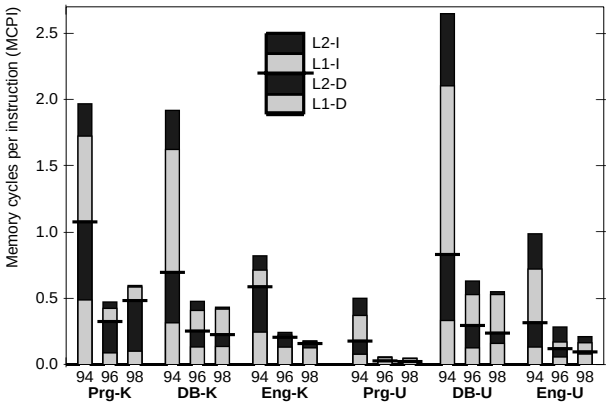

FIGURE 4.4. Memory stall cycles per instruction.
This figure shows the memory stall cycles per instruction (MCPI) for the three machine models running the three workloads. MCPI is broken down by its source: level-1 and level-2 data cache stall (L1-D & L2-D), level-1 and level-2 instruction cache stall (L1-I & L2-I). Results are presented for both non-idle kernel (-K) and user (-U) execution.

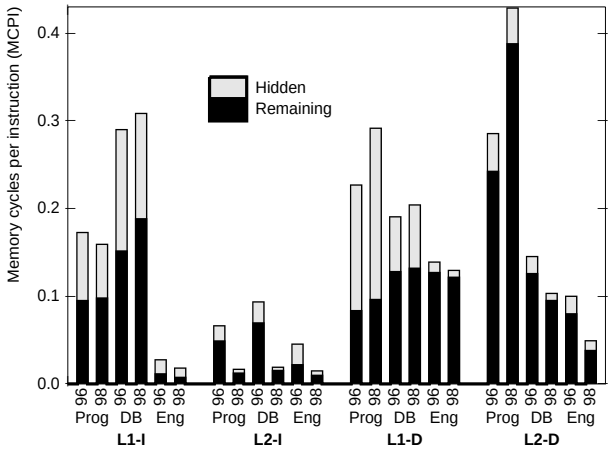

FIGURE 4.5. Kernel stall time hidden by the 1996 and 1998 models.
This figure shows the non-idle kernel MCPI of the dynamically scheduled 1996 and 1998 models and the part of the MCPI which is hidden. The results for the level-1 instruction miss stall (L1-I), level-2 instruction miss stall (L2-I) and level-1 (L1-D) and level-2 (L2-D) data stalls are shown.

processors. This results in a negligible user-level MCPI, especially when compared to the kernel's memory system behavior. In contrast, the engineering and database workloads consist of very large programs which continue to suffer from significant memory stall time. In fact, their memory system behavior is quite comparable to that of the kernel. Other studies have concluded that the memory system behavior of the kernel was worse than that of application programs [3]. We find that this is true for smaller programs, but does not hold for large applications. The implication is that processor improvements targeted at large applications will likely benefit kernel performance as well.

The improved memory system behavior of the 1996 and 1998 models is due to two features: latency tolerance and larger caches. In Section 4.3 and Section 4.4 we examine separately the benefits of these features.

4.3 Latency Tolerance

Dynamically scheduled processors can hide portions of cache miss latencies. Figure 4.5 presents the amount of memory stall time observed in the 1996 and 1998 models and compares it with the memory stall time of the comparable statically-scheduled models. The numbers for this figure were computed by running the 1994 model configured with the same caches and memory system as the next-generation models and comparing the amount of memory stall seen by the processor.

The figure emphasizes two results. First, dynamically scheduled processors are more effective at hiding the shorter latencies of level-1 misses than that of level-2 misses. Dynamically scheduled processors hide approximately half of the latency of kernel level-1 misses. The notable exception is the engineering workload which spends most of its limited kernel time in the UTLB miss handler. We discuss the special behavior of this routine in Section 4.5.

Unfortunately, level-2 caches do not benefit from latency hiding as much as level-1 caches. The longer latency of a level-2 miss makes it more difficult for dynamic scheduling to overlap significant portions of the stall time with execution. Level-2 miss costs are equivalent to the cost of executing hundreds of instructions. There is simply not enough instruction window capacity to hold the number of instructions needed to overlap this cost. Although it is possible to overlap level-2 stall with other memory system stall, we didn't observe multiple outstanding level-2 misses frequently enough to significantly reduce the stall time.

A second and somewhat surprising result from Figure 4.5 is that the future processor models are particularly effective at hiding the latency of instruction cache misses. This is non-intuitive because when the instruction fetch unit of the processor stalls on a cache miss, it can no longer feed the execution unit with instructions. The effectiveness is due to the decoupling of the fetch unit from the execution unit. The execution unit can continue executing the instructions already in the window while the fetch unit is stalled on an instruction cache miss. Frequent data cache misses cause both the executing instruction and dependent instructions to stall, and give the instruction unit time to get ahead of the execution unit. Thus, next generation processors overlap instruction cache miss latency with the latency of data cache misses while statically-scheduled processors must suffer these misses serially.

4.4 Larger Cache Sizes

Future processors will not only have latency tolerating features, but will also have room for larger caches. The sizes of the caches are controlled by a number of factors including semiconductor technology as well as target cycle time. We first examine sizes likely to be found in on-chip level-1 caches and then explore the sizes likely to be found in off-chip level-2 caches.

4.4.1 Level-1 Cache

Figure 4.6 presents the average number of cache misses per instruction for each of the workloads. We explore a range of sizes that could appear in level-1 caches of future processors. We model separate data and instruction caches.

One of the key questions is whether increasing cache sizes will reduce memory stall time to the point where operating system developers do not need to worry about it. The miss rates in Figure 4.6 translate into different amounts of memory stalls on different processor models. For example, the maximum cost of a level-1 cache miss which hits in the level-2 cache on the 1998 model is 60 instructions. A miss rate of just 0.4% on both instruction and data level-1 caches means that the processor could spend half as much time stalled as executing instructions. This can be seen in the memory stall time on the 1998 model.

Since larger caches will not avoid all misses, we next classify them into 5 categories based on the cause of a line's replacement. *Cold* misses occur on the first reference to a cache line. *KERN-self* occur when the kernel knocks its own lines out of the cache and

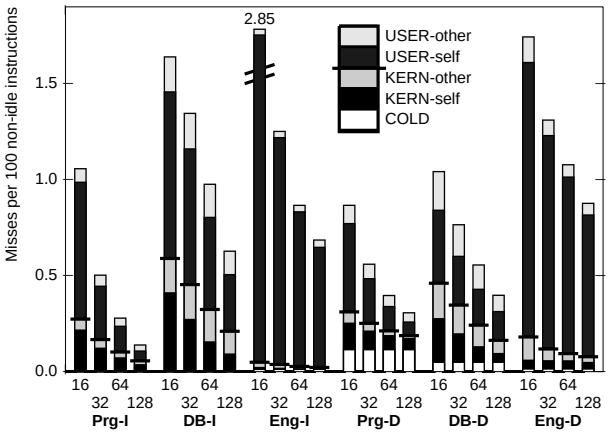

FIGURE 4.6. Cache misses for several I- and D-cache sizes.
All caches are two-way set-associative with LRU replacement. The instruction caches (-I) have a cache line size of 64 bytes, and the data caches (-D) have 32 byte lines. Misses are broken down by mode (user, kernel) and by the cause of the miss. All cache sizes are in kilobytes.

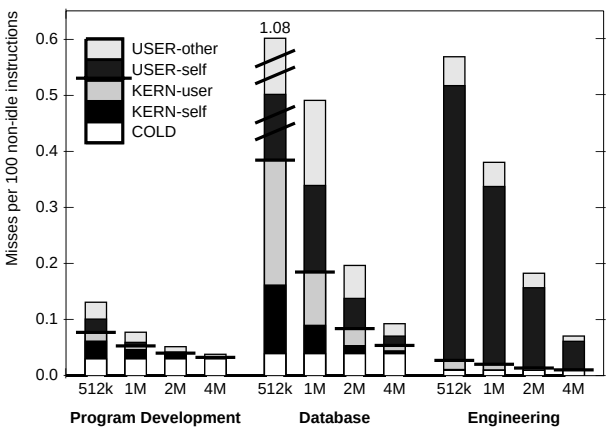

FIGURE 4.7. Behavior of unified level-2 caches.
All caches are 2-way associative with LRU replacement and have 128 byte lines. As in Figure 4.6, we break the misses down by type.

KERN-other misses occur when a user process replaces the kernel's cache lines. *USER-self* misses occur when a user process knocks its own lines out of the cache and *USER-other* misses occur when a cache line is replaced by the kernel or by a different user process.

Figure 4.6 shows that larger caches are more effective at removing self-inflicted misses in both the user applications (USER-self) and the kernel (KERN-self) than they are at reducing interference misses (USER-other and KERN-other). This is most striking in the database workload where USER-self and KERN-self misses steadily decline with larger caches while the USER-other and KERN-other misses remain relatively constant.

Apart from techniques such as virtual memory page coloring, the operating system has no control over user-self cache misses. Any improvements will necessarily have to come from improved memory systems. However, operating system designers can address KERN-self cache misses. For example, recent work has shown how code re-organization can reduce these instruction misses [14].

Reducing the KERN-other and the USER-other misses is more problematic. Most of the USER-other misses are due to interference between the user and kernel rather than interference between two user processes. In fact, these two miss types are quite complementary. When the kernel knocks a user line from the cache, the user often returns the favor knocking a kernel line out of the cache. Although re-organizing the kernel to reduce its cache footprint could decrease the amount of interference, the problem will remain. As long as two large code segments are trying to share the same level-1 instruction cache, there will be potential for conflicts.

We also explored the impact of cache associativity by looking at both direct-mapped and 4-way associative caches. Like cache size increases, higher associativities reduce self-induced misses significantly more than interference misses. We hypothesize that the entire state in the relatively small level-1 caches is quickly replaced after each transition between kernel and user mode. As long as this is the case, associativity will not significantly reduce interference misses.

4.4.2 Level-2 Caches

Figure 4.7 presents the miss rates for cache sizes likely to be found in off-chip, level-2 caches. These caches typically contain both instructions and data, have larger line sizes, and incur significantly higher miss costs than on-chip caches. For example, the latency of a level-2 cache miss in the 1998 model is equivalent to the execution of 500 instructions. A cache miss rate of just 0.1% could stall the 1998 processor for half as much time as it spends executing instructions. The smallest miss rate shown in Figure 4.7 would slow down such a processor by as much as 25%.

Similarly to level-1 caches, larger level-2 caches reduce misses substantially, but still do not totally eliminate them. For the program development and database workloads, a 4MB level-2 cache eliminates most capacity misses. The remaining misses are cold misses[1].

4.5 Characterization of Operating System Services

In previous sections we have looked at kernel behavior at a coarse level, focusing on average memory stall time. In order to identify the specific kernel services responsible for this behavior, we use SimOS annotations to decompose the kernel time into the services that the operating system provides to user processes. Table 4.8 decomposes operating system execution time into the most significant operating system services.

One common characteristic of these services is that the execution time breakdown does not change significantly when moving from the 1994 to the 1996 and 1998 models. This is encouraging since optimizations intended to speed up specific services today will likely be applicable on future systems. We now examine separately the detailed operating system behavior of the three workloads.

Program development workload. On the 1994 model, this workload spends about 50% of its kernel time performing services related to the virtual memory system calls, 30% in file system related services, and most of the remaining time in process management services. Memory stall time is concentrated in routines that access large blocks of memory. These include `DEMAND-ZERO` and copy-on-write (`COW`) faults processing as well as the `read` and `write` system calls that transfer data between the application's address space and the kernel's file cache.

The larger caches of the 1998 model remove most of the level-2 capacity misses of the workload. The remaining stall time is due mostly to level-1 cache misses and to level-2 cold misses.

1. Most of the cold misses are really misclassified capacity misses. The reason that they are misclassified is that the initial accesses to the memory preceded the detailed simulation start. Had the detailed examination started at boot time, these cold misses would have been classified as capacity misses.

	Service	%Kernel			Computation			1994 configuration					1998 configuration					Latency			Events per Second	%Cold misses
		1994	1996	1998	Avg(94) [microsec]	Speedup-96	Speedup-98	CPI	i-MCPI	d-MCPI	%stall (I)	%stall (D)	CPI	i-MCPI	d-MCPI	%stall (I)	%stall (D)	Avg(94) [microsec]	Speedup-96	Speedup-98		
PROGRAM DEVELOPMENT	DEMAND ZERO	18.1	17.1	25.0	51	3.3	5.1	3.75	0.52	2.23	4%	16%	1.84	0.05	1.43	1%	35%	51	3.3	5.1	532	54
	QUICK FAULT	10.5	9.9	7.7	13	3.3	9.6	2.09	0.80	0.30	6%	2%	0.55	0.04	0.05	1%	1%	13	3.4	9.6	1224	0
	execve	10.2	10.7	10.5	1180	3.0	6.9	2.64	0.89	0.75	5%	4%	0.95	0.12	0.38	2%	7%	1221	2.5	6.6	13	8
	write	10.0	9.4	10.4	47	3.3	6.8	2.97	0.73	1.24	4%	6%	1.10	0.09	0.60	2%	10%	52	3.4	7.1	320	12
	read	9.1	8.5	7.8	87	3.4	8.2	3.52	0.98	1.54	4%	6%	1.07	0.10	0.53	1%	7%	87	3.3	8.2	157	4
	C.O.W. FAULT	6.3	5.5	6.4	82	3.6	6.9	4.18	0.48	2.70	1%	6%	1.53	0.04	1.10	0%	8%	82	3.6	6.9	115	12
	UTLB	4.7	8.4	7.0	0.08	1.7	4.6	1.77	0.04	0.75	0%	3%	0.96	0.00	0.07	0%	1%	0.08	1.7	4.6	87442	0
	open	4.6	4.7	3.8	100	3.1	8.4	3.27	1.73	0.54	4%	1%	0.96	0.28	0.18	2%	1%	2594	0.9	1.0	69	0
	fork	4.2	4.3	4.5	568	3.0	6.5	3.03	0.76	1.26	2%	3%	1.17	0.12	0.62	1%	4%	35 ms	3.8	8.4	11	5
	exit	3.5	4.2	3.4	481	2.6	7.4	2.37	0.66	0.71	1%	2%	0.80	0.11	0.23	1%	2%	482	2.6	7.4	11	0
	brk	2.8	2.7	2.2	33	3.3	9.1	3.80	2.06	0.74	2%	1%	1.04	0.37	0.17	1%	1%	33	3.4	9.1	130	0
	PFAULT	2.4	2.4	1.8	14	3.2	9.2	2.32	1.07	0.26	2%	0%	0.64	0.12	0.05	1%	0%	14	3.2	9.2	266	0
	CLOCK INT	2.4	1.0	0.5	36	3.8	12.5	6.21	3.57	1.65	2%	1%	1.70	0.63	0.42	0%	0%	37	3.8	12.3	100	0
	close	1.9	1.9	1.5	31	3.2	9.1	3.63	2.02	0.60	2%	0%	0.99	0.30	0.17	1%	0%	916	1.2	1.2	92	0
	unlink	1.5	1.4	1.2	197	3.2	8.8	3.56	1.84	0.72	1%	0%	1.00	0.26	0.22	1%	0%	17 ms	1.0	1.0	11	0
	Other	7.8	7.8	6.3	--	--	--	--	--	--	6%	2%	--	--	--	3%	2%	--	--	--	--	1
DATABASE	read	15.2	12.4	12.6	80	3.6	9.4	3.94	2.11	0.83	12%	5%	1.07	0.29	0.28	8%	7%	2090	1.2	1.3	466	30
	select	15.2	13.8	13.5	86	3.1	8.4	2.80	0.99	0.81	8%	6%	0.90	0.17	0.26	6%	9%	44 ms	1.2	1.2	431	0
	UTLB	12.9	26.0	26.2	0.07	1.4	3.8	1.35	0.00	0.35	0%	5%	0.90	0.00	0.08	0%	5%	0.07	1.4	3.8	424582	0
	write	8.4	7.0	6.8	88	3.3	8.8	4.20	2.52	0.68	7%	2%	1.28	0.51	0.21	6%	2%	734	1.7	2.9	233	0
	ioctl	8.1	6.3	5.8	37	3.4	9.3	3.87	1.83	1.04	6%	3%	1.05	0.27	0.26	3%	3%	5309	1.1	1.0	534	0
	SW INT	7.2	5.7	5.4	62	3.7	10.3	5.26	2.93	1.33	6%	3%	1.30	0.44	0.30	4%	3%	62	3.7	10.3	283	0
	send	4.3	3.7	3.3	76	3.4	10.1	5.31	3.30	1.01	4%	1%	1.31	0.54	0.25	3%	1%	138	3.5	10.2	138	0
	DEMAND ZERO	4.1	3.7	5.8	69	3.2	5.5	4.88	1.37	2.52	2%	3%	2.24	0.27	1.54	2%	9%	70	3.2	5.5	145	53
	DISK INT	3.4	2.6	2.1	28	3.6	11.4	4.58	2.70	0.88	3%	1%	1.02	0.32	0.18	2%	1%	28	3.6	11.4	289	0
	DBL_FAULT	3.0	1.9	1.5	0.87	1.5	4.9	1.72	0.48	0.22	1%	1%	0.87	0.10	0.17	0%	1%	0.88	1.5	4.9	8553	0
	fcntl	2.5	2.3	2.0	22	3.1	9.1	5.65	3.13	1.52	2%	1%	1.56	0.68	0.37	2%	1%	22	3.1	9.1	286	0
	syscall	2.0	1.9	2.0	17	2.8	7.1	2.47	1.04	0.43	1%	1%	0.87	0.26	0.14	1%	1%	17	2.8	7.2	286	0
	recv	1.8	1.5	1.3	31	3.4	10.4	3.42	1.96	0.45	1%	0%	0.82	0.23	0.11	1%	0%	31	3.4	10.5	140	0
	gettimeofday	1.4	1.5	1.5	9	2.7	7.2	3.29	1.77	0.52	1%	0%	1.15	0.25	0.14	1%	0%	9	2.8	7.3	391	0
	sigprocmask	1.2	1.3	1.3	5	2.8	6.8	2.02	0.94	0.08	1%	0%	0.74	0.19	0.03	1%	0%	5	2.8	6.8	571	0
	Other	9.2	8.5	8.9	--	--	--	--	--	--	6%	3%	--	--	--	5%	5%	--	--	--	--	12
ENGINEERING	UTLB	69.8	86.1	87.4	0.07	1.4	3.6	1.49	0.01	0.47	1%	33%	1.03	0.00	0.14	0%	41%	0.07	1.4	3.6	485840	2
	CLOCK INT	12.5	2.7	0.9	58	2.8	8.4	9.18	4.86	3.33	10%	7%	2.69	0.97	1.06	1%	1%	59	2.8	8.4	100	1
	DBL_FAULT	6.0	5.0	4.7	0.83	2.7	7.6	1.89	0.44	0.41	2%	2%	0.62	0.01	0.01	0%	0%	0.83	2.7	7.6	3365	0
	exit	2.9	2.1	2.9	1099	2.3	4.4	2.27	0.34	0.93	1%	2%	1.28	0.12	0.64	1%	5%	1099	2.2	4.4	1	29
	DEMAND ZERO	1.7	0.9	1.2	89	3.3	6.6	6.46	2.50	2.95	1%	1%	2.41	0.33	1.66	1%	3%	89	3.3	6.6	9	16
	SW INT	1.4	0.5	0.4	44	2.4	5.5	3.87	1.94	0.96	1%	0%	1.08	0.31	0.28	0%	0%	74	2.0	4.2	14	2
	DAEMONS	1.1	0.6	0.4	273	2.5	2.9	7.89	1.39	5.51	0%	1%	7.94	0.74	6.69	0%	1%	273	2.5	2.9	2	28
	Other	4.6	2.1	2.1	--	--	--	--	--	--	3%	1%	--	--	--	2%	2%	--	--	--	--	16

TABLE 4.8. Detailed breakdown of the uniprocessor workloads kernel activity.

The most significant services of the uniprocessor workloads are presented in order of their fraction of kernel computation time on the 1994 machine model. Lower-case services denote UNIX system calls, and upper-case services denote traps and interrupts(INT). `UTLB` reloads the TLB for user addresses. A `DBL_FAULT` occurs when the UTLB takes a TLB miss. A `QUICK FAULT` is a page fault where the page is already in memory, and `PFAULT` denotes protection violation traps used for modify bit emulation and other purposes. We compare the memory system behavior of the 1994 and 1998 models. The cycles per instruction (CPI) as well as the instruction and data memory CPI (i-MCPI, d-MCPI), are indicators of processor performance. The concentration of instruction and data memory stall (%stall (I) and %stall (D)) is expressed in percentage of all kernel memory stalls. Latency numbers can be compared to computation time to determine the importance of I/O and scheduling on the latency of various services.

The `DEMAND-ZERO` page fault generates more than 50% of all cold misses. This service amounts to 18% of kernel execution time in the 1994 model and 25% of kernel execution time in the 1998 model. In general, the services with large concentrations of cold misses, including the copy-on-write fault and the `write` system call, increase in importance. Fortunately, the operating system can address this problem. Many of the cold misses are due to the virtual memory page allocation policy which does not re-use recently freed pages that would potentially still reside in the level-2 cache. The workload touches a total of 33 megabytes of memory. Given the large number of process creations and deletions, it is likely that this memory footprint could be reduced by modifying the page allocator to return recently freed pages (the current allocation policy delays page reuse to deal with instruction cache coherency issues).

In contrast, the latency tolerating features work well on the block copy/zero routines. Non-blocking caches allow the processor to overlap cache miss stall with the block movement instructions. Additionally, they permit multiple outstanding cache misses, allowing stall times to overlap. These features effectively hide all level-1 block-copy stall time that is present in the 1994 model. This effect can be seen in the relatively large speedups obtained by the `read` and `write` system calls, and the large level-1 data cache stall time hidden in the workload (see Prog L1-D in Figure 4.5).

The operating system can still potentially improve the performance of level-2 cache behavior in the block copy/zero routines. The current routines fill the load/store buffer of the processor before they are able to generate multiple outstanding level-2 cache misses. By re-coding the functions to force cache misses out ear-

lier, we can overlap the stall time of multiple misses, substantially reducing the total level-2 cache stall time. Re-coding this type of routine to take advantage of next generation processors can reduce the performance impact of the block-copy routines.

Database workload. This workload spends over half of its kernel time in routines supporting inter-process communication between the client and the database server and about one third of its time performing disk I/Os for the database server. Additionally, the 1994 model spends 13% of the kernel time servicing UTLB faults. Unlike the program development workload, the memory stall of this workload is evenly spread among the kernel services. Kernel instruction cache performance is particularly bad in the 1994 model, with instruction MCPIs of over 2.0 for several of the major services.

Most services encounter impressive speedups on the 1996 and 1998 models. The improvement in the cache hierarchy dramatically reduces the instruction and data MCPI of the main services. Unfortunately, one of the major services only shows a moderate speedup: the UTLB miss handler is only 1.4x (1996) and 3.8x (1998) faster than the 1994 model. Because of this lack of speedup the time spent in the UTLB handler increases to a quarter of the kernel time (10% of the non-idle execution time) in the 1998 model. The UTLB handler is a highly-tuned sequence of 8 instructions that are dependent on each other. They do not benefit from the multiple issue capabilities of the 1996 and 1998 models. Performance improvements will need to come from a reduction of the TLB miss rate. This can be achieved by improved TLB hardware or through the use of larger or variable-sized pages

Engineering workload. This workload makes relatively few direct requests of the operating system, and the kernel time is dominated by the UTLB handler and by clock interrupt processing (`CLOCK INT`). The UTLB miss handler has the same behavior as in the database workload. Fortunately, it accounts for only 7% of the total execution time. The importance of the clock interrupt handler diminishes significantly on the 1996 and 1998 models. The service is invoked fewer times during the workload and larger caches can retain more of its state between invocations.

Latency effects. We have discussed the impact of architectural trends on the computation time of specific services. We now focus on their latency.

Table 4.8 also contains the request rate and the average request latency of the services. The average latency is an interesting metric as user processes are blocked during the processing of that service. For most services, the computation time is equivalent to the latency. However, the computation time is only a small fraction of the latency of some services. These services are either involved in I/O, blocked waiting for an event, or descheduled.

Services such as the open, close, and unlink system calls of the program development workload and the read and write system calls of the database workload frequently result in disk I/Os. These system calls are limited by the speed of the disks on all processor models, resulting in both long delays for the calling process and show very little, if any, speedup. Only changes in the file system will reduce these latencies. System calls which block on certain events, such as select, also experience longer latencies. The long latency of the fork system call results from the child process getting scheduled before the forking parent is allowed to continue.

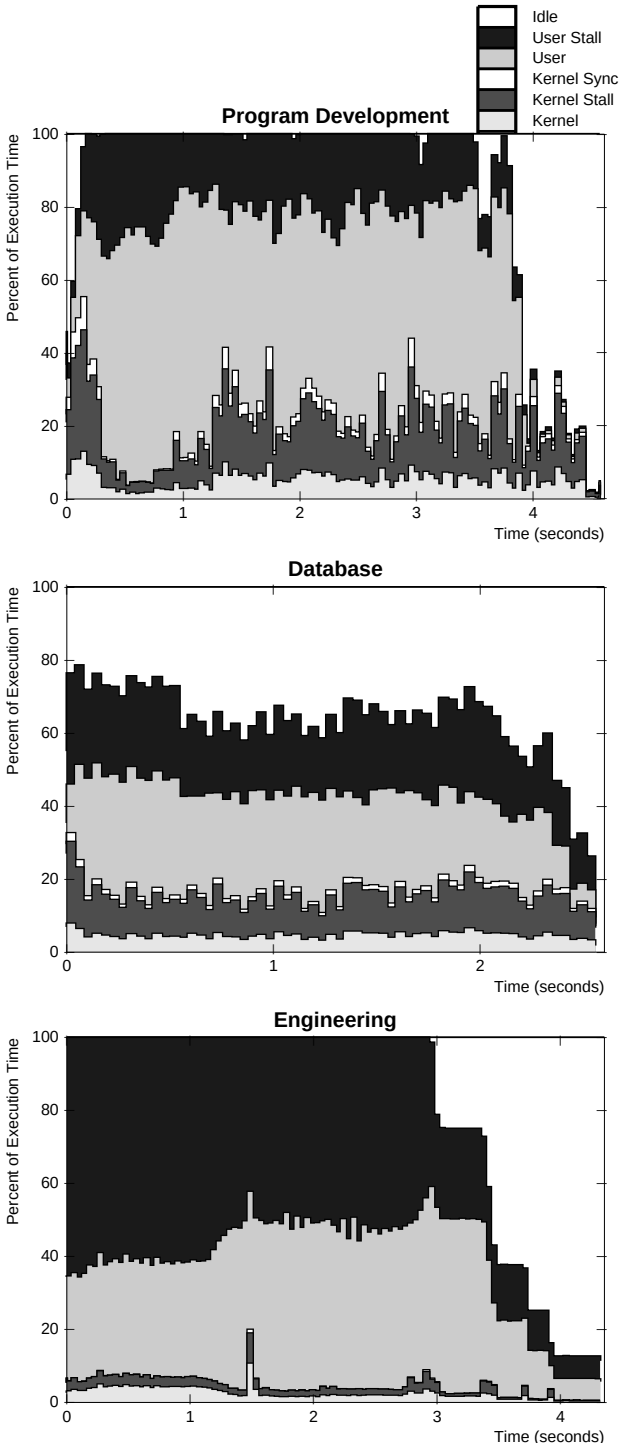

FIGURE 5.1. Profiles of multiprocessor workloads.
The execution time of each workload is separated into the time spent in user, kernel, sync, and idle modes on the eight-CPU model. User and kernel modes are further subdivided into instruction execution and memory stall.

5 Multiprocessor Effects

The move to small-scale shared-memory multiprocessors appears to be another architectural trend, with all major computer vendors developing and offering such products. To evaluate the effects of this trend, we compare the behavior of the kernel on a uniprocessor to its behavior on an 8-CPU multiprocessor. Both configurations use the 1994 CPU model and memory hierarchy. The multiprocessor workloads are scaled-up versions of the uniprocessor ones, as described in Section 2.2. The multiprocessor version of the IRIX kernel is based on the uniprocessor version and includes the requisite locking and synchronization code. It is

OS events	Prg-Dev	Database	Eng
Duration	4.7 secs	2.6 secs	4.4 secs
Process creations	80	1.1	< 1
Context switches	1633	3929	193
Interrupts	1428	1853	843
System calls	8246	13546	51
TLB refills	655 x 10^3	1374 x 10^3	2653 x 10^3
VM faults	17166	30309	30481
Other exceptions	3216	3097	118
Hardware events			
Instructions	1105 x 10^6	1074 x 10^6	882 x 10^3
L1-I cache misses	18359 x 10^3	14961 x 10^3	25229 x 10^3
L1-D cache misses	10481 x 10^3	7124 x 10^3	10017 x 10^3
L2-cache misses	1948 x 10^3	2500 x 10^3	3358 x 10^3
Disk I/Os	214	829	1

TABLE 5.2. Event rates for the multiprocessor workloads.
Results are aggregated over all eight processors. All rates are reported as events per second.

designed to run efficiently on the SGI Challenge series of multiprocessors, which supports up to 36 R4400 processors.

5.1 Base characterization

We begin our investigation with a high-level characterization of the multiprocessor workloads. Table 5.2 presents hardware and software event rates for the workloads (aggregated over all processors), and Figure 5.1 presents the execution breakdown over time. While the uniprocessor and multiprocessor workloads have different compositions (also see Figure 4.1 and Table 4.2), the workloads are scaled to represent a realistic load of the same application mix running on the two configurations. The workloads drive the operating system in similar ways, and thus provide a reasonable basis for performance comparisons.

Compared to Figure 4.1, there is an increase in idle time. This idle time is due to load imbalance towards the end of the program-development and engineering workloads, and due to I/O bottlenecks for the database workload. Two of the three workloads show an increase in the relative importance of kernel time. The kernel component increases due to worse memory system behavior and synchronization overheads. The portion of non-idle execution time spent in the kernel in the program development workload rises from 16% to 24%, and in the engineering workload it rises from 4.6% to 5.3%. The database workload interestingly shows the opposite behavior. Although the fraction of time spent in the kernel decreases from 38.2% to 24.9% on the multiprocessor, this reduction is not due to improved kernel behavior. Rather, the multiprocessor version of the database server is a parallel application and requires more computation per transaction.

5.2 Multiprocessor Overheads

There are a number of overheads found in multiprocessor system that are not present in uniprocessors. This section examines two of these overheads: synchronization and additional memory-stalls.

Synchronization. The multiprocessor IRIX kernel uses spinlocks to synchronize access to shared data structures. Overheads include the time to grab and release locks, as well as the time spent waiting for contended locks. Spinlocks are not used in the uniprocessor version of the kernel.

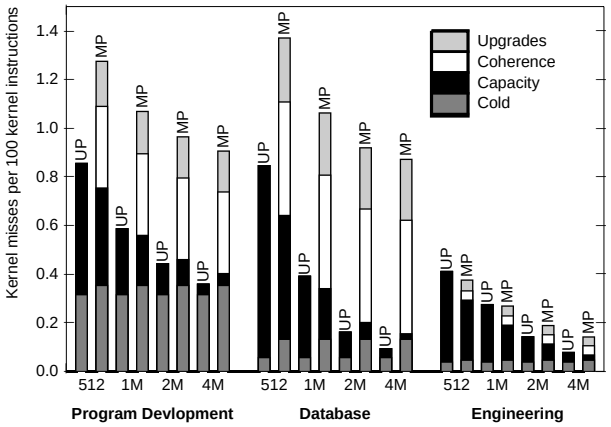

FIGURE 5.3. Level-2 cache miss rates in kernel mode.
This compares uniprocessor and multiprocessor miss rates in kernel mode for a range of cache sizes. We model a unified level-2 cache with 128 byte lines. Misses are classified as cold, capacity, coherence, or upgrades.

The importance of synchronization time varies greatly with the workload. Synchronization time accounts for 11.2%, 7.6%, and 1.4% of kernel execution time in the 8-CPU program-development, database, and engineering workloads respectively. To better understand how this time will scale with more processors, we examine synchronization behavior for individual system calls in Section 5.3.

Memory stall time. For the multiprocessor configuration, SimOS models an invalidation-based cache-coherence protocol. Cache-coherence induces two new types of memory stalls that do not occur on uniprocessors. A *coherence miss* occurs because the cache line was invalidated from the requestor's cache by a write from another processor.[1] An *upgrade stall* occurs when a processor writes to a cache line for which it does not have exclusive ownership. The upgrade requires communication to notify other processors to invalidate their copy of the cache line.

Figure 5.3 compares the uniprocessor and multiprocessor kernel miss rates for a range of level-2 cache sizes. In contrast to uniprocessors, larger caches do not reduce the miss rate as dramatically. The reason is simple; coherence misses do not decrease with increasing cache size. Coherence misses correspond to communication between processors and are oblivious to changes in cache size.

The implications of this observation are quite serious. In uniprocessors, larger caches significantly reduce the miss rates, allowing large performance gains in the 1996 and 1998 models. In multiprocessors, larger caches do not reduce the miss-rate as effectively, and we will see a much higher stall time in future machines.

Although, we did not simulate a multiprocessor machine with the next generation CPUs, it is possible to make rough estimates regarding the magnitude of the problem. As mentioned in Section 4.4.2, a level-2 cache miss rate of just 0.1% stalls the 1998 processor for half as much time as it spends executing instructions. Figure 5.3 shows that for the program development and database workloads we will have at least 0.8 misses per 100 kernel instructions. Thus, although the memory stall times for the 1994 multiprocessor do not look too bad, the stall times for future machines will be much worse. In the next section we examine specific operating system services and suggest possible improvements for

1. Cache-coherent DMA causes coherence misses in the uniprocessor workloads, but they constitute a very small fraction of the total misses.

	Service	%Kernel		Computation		Computation breakdown (% Time)						Latency				Events per second	Lock acquires per call	Coherence misses per call
		MP (1994)	UP (1994)	Avg-MP [microsec]	Slowdown over UP	% Instructions	% Spinlock	% Coherence	% Upgrades	% I-cache	% D-cache (cold+cap)	Avg-MP [microsec]	Slowdown over UP	Avg-Blocked [microsec]	Avg-Descheduled [microsec]			
PROGRAM DEVELOPMENT	DEMAND FILL	15.5	18.1	65	28%	22	10	5	3	13	47	65	29%	--	--	3863	8	5
	QUICK FAULT	10.8	10.5	20	53%	33	17	8	3	30	9	20	54%	--	0.24	8879	5	2
	execve	10.4	10.2	1799	52%	27	13	10	6	25	20	2241	84%	203	239	94	572	337
	write	9.0	10.0	63	34%	29	8	5	3	23	32	75	44%	4	8	2339	13	6
	PFAULT	7.8	2.4	59	340%	19	13	13	6	30	19	447	3198%	36	352	2126	13	14
	read	7.0	9.1	101	17%	24	5	5	2	25	39	131	50%	0.36	29	1131	10	10
	fork	4.9	4.2	998	76%	21	15	12	7	19	26	18 ms	-48%	27	17 ms	80	281	239
	open	4.9	4.6	158	58%	23	9	11	5	40	12	2059	-21%	1462	439	503	51	33
	exit	4.6	3.5	927	93%	23	18	18	11	15	14	1322	174%	177	218	80	345	335
	C.O.W. FAULT	4.4	6.3	109	33%	19	7	4	2	10	58	109	33%	--	--	658	11	8
	UTLB	3.6	4.7	0.09	10%	62	0	2	0	13	24	0.09	10%	--	--	654603	0	0
	brk	3.5	2.8	60	84%	16	15	13	2	33	20	61	85%	--	0.53	944	18	15
	close	2.0	1.9	49	56%	19	12	12	5	41	11	684	-25%	492	144	657	14	11
	CLOCK INT	1.9	2.4	39	8%	13	3	11	7	50	17	39	6%	--	--	798	1	8
	unlink	1.6	1.5	323	64%	19	10	15	7	38	12	14 ms	-16%	12 ms	2491	81	86	92
	Other	8.1	7.8	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
DATABASE	select	12.4	15.2	154	79%	33	18	12	4	18	15	47 ms	6%	0.10	47 ms	1088	97	34
	read	12.2	15.2	121	51%	19	7	13	8	37	16	4217	102%	4051	45	1363	23	31
	UTLB	10.2	12.9	0.10	34%	70	0	0	0	3	27	0.10	34%	--	--	1373819	0	0
	ioctl	8.0	8.1	66	76%	16	7	16	7	39	14	8281	56%	0.29	8215	1644	13	20
	QUICK FAULT	8.0	0.3	54	273%	16	9	12	6	41	16	65	344%	4	7	1981	9	13
	write	7.5	8.4	144	63%	17	8	15	7	43	11	795	8%	608	44	705	27	40
	rtnetd	5.7	0.0	196	0%	18	10	18	8	29	17	198	0%	--	2	395	43	69
	END_IDLE	5.0	0.6	16	173%	16	4	27	18	26	10	16	147%	--	--	4054	0	8
	syscall	3.9	2.0	64	273%	35	2	22	0	11	30	65	274%	--	0.77	828	3	28
	DISK INT	3.2	3.4	48	69%	17	6	15	7	47	8	48	69%	--	--	891	7	14
	PFAULT	3.2	0.0	24	69%	27	10	12	5	37	9	32	124%	3	5	1784	7	5
	send	2.5	4.3	83	9%	19	8	11	5	46	12	84	-39%	--	0.71	410	16	17
	DBL_FAULT	2.4	3.0	1	38%	44	0	1	0	38	16	1	37%	--	--	26894	0	0
	exit	2.0	0.9	1136	78%	27	12	10	11	14	25	1146	73%	2	8	24	481	231
	CLOCK INT	1.8	0.9	30	33%	13	3	11	7	52	14	30	33%	--	--	798	1	6
	fcntl	1.6	2.5	26	19%	15	4	9	3	51	17	26	20%	--	0.44	828	4	4
	recv	1.4	1.8	42	36%	23	4	7	4	50	12	43	37%	--	0.49	436	9	5
	Other	9.0	20.5	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--
ENGINEERING	UTLB	55.8	69.8	0.07	8%	76	0	0	0	0	23	0.07	8%	--	--	2652699	0	0
	CLOCK INT	12.9	12.5	55	-5%	10	2	8	7	53	21	55	-6%	--	--	799	2	8
	DBL_FAULT	10.6	6.0	1	55%	36	0	0	2	51	11	1	55%	--	--	28062	0	0
	QUICK FAULT	8.0	0.7	12	-57%	41	5	1	3	34	16	19	-29%	5	3	2361	4	0
	exit	6.6	2.9	5290	381%	30	6	7	17	4	35	5994	445%	--	704	4	2085	780
	DAEMONS	1.5	1.1	510	87%	34	3	2	7	8	47	512	87%	--	1	10	48	17
	DEMAND FILL	1.2	1.7	110	23%	14	3	3	4	34	42	110	23%	--	--	38	8	6
	Other	3.5	5.3	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

TABLE 5.4. Detailed breakdown of the multiprocessor workloads' kernel activity.
The most significant services of the multiprocessor workloads are presented in order of their importance along with several statistics. Again, lower-case services denote UNIX system calls. Uppercase services are as described in Table 4.8. We compare the average computation time of each service to its execution on the 1994 uniprocessor system. For each service, we indicate the fraction of computation time spent in execution, synchronization, suffering coherence misses, requesting upgrades or memory stalls. Synchronization, coherence misses and upgrades are overheads that are inherent to multiprocessors and not present in uniprocessors. The latency of each service is broken down into average computation time, blocked or I/O latency, and scheduling latency. Coherence misses that occur in the synchronization routines are not part of the coherence columns but are factored in the synchronization categories.

reducing the number of coherence misses.

5.3 Characterization of Operating System Services

To better understand the effects of multiprocessor overheads on kernel performance, Table 5.4 compares the performance of multiprocessor operating system services to their uniprocessor counterparts. With a few notable exceptions, the relative importance of each service remains the same. This implies that most services suffer similar slowdowns in the transition to multiprocessors.

Program development workload. The top four services of the program development workload, `DEMAND-ZERO`, `QUICK-FAULT`, `execve`, and `write`, account for about 45% of the kernel execution time. These services suffer a slowdown of between 30% and 50% compared to their uniprocessor equivalents. Spinlock synchronization accounts from 8% to 17% the execution time of these services. More than half of synchronization overhead is due to contention on a single lock (`memlock`) that protects the data structures that manage the physical memory of the machine.

Coherence misses and upgrades comprise between 8% and 19% of the time. Coherence misses and upgrades represent com-

munication between processors on shared-memory multiprocessors. Unfortunately, some of the coherence misses are caused by false sharing. False sharing occurs when unrelated variables reside on the same cache line. Whenever one of these variables is written, the entire cache line must be invalidated from other processors' caches. As a result, subsequent reads to any variable on the cache line will miss. One extreme example of false sharing involves `memlock`. The cache line containing `memlock` also contains 22 other variables that the compiler and linker happen to allocate adjacently. This line alone accounts for 18% of all coherence misses in the kernel. As the relative cost of coherence misses increases, programmers and compilers will have to pay much more attention to this type of data layout problem.

Table 5.4 also compares the latency of the services on both platforms. Unlike the comparison of computation time, which always reports a slowdown, some services actually have a shorter latency on multiprocessors. The `fork` system calls return in half the uniprocessor time because of the presence of alternate CPUs to run concurrently both the forking parent and the child. On a uniprocessor, the parent gets preempted by the newly created child process. System calls that perform I/O such as `open`, `close`, and `unlink` also show speedups of 15% to 25% over the uniprocessor run. This is not due to a reduction in I/O latency but again due to the increased probability that a CPU is available when a disk I/O finishes. More specifically, the IRIX scheduler does not preempt the currently running process to reschedule the process for which an I/O finishes, and this causes the uniprocessor latency to be longer than simply the disk I/O latency. This scheduling policy also increases the latency of functions that synchronize with blocking locks. This can be seen in the 32-fold slowdown of the `PFAULT` exception.

Database workload. The general trends for the database workload look similar to those in the program development workload. The fraction of computation time taken by key system calls remains the same across uniprocessor and multiprocessor implementations. However, several aspects are unique to this workload. The database workload heavily utilizes inter-process communication, which is implemented differently by the uniprocessor and multiprocessor kernels. The uniprocessor kernel implements the socket `send` system call by setting a software interrupt (`SW INT` in Table 4.8) to handle the reception of the message. The multiprocessor version hands off the same processing to a kernel daemon process (`rtnetd`). The advantage of this latter approach is that the daemon process can be scheduled on another idle processor. As Table 5.4 shows, this reduces the latency of a `send` system call by 39% on the multiprocessor version.

Another significant difference is the increased importance of the `END_IDLE` state which takes 0.6% of kernel time on the uniprocessor but 5.0% of the time on the multiprocessor. This state captures the time spent between the end of the idle loop and the resumption of the process in its normal context. Two factors explain this difference. First, in the multiprocessor, all idle processors detect the addition of a process to the global run queue, but only one ends up running it. The rest (approximately one quarter of the processors in this workload) return back to the idle loop, having spent time in the `END_IDLE` state. Second, a process that gets rescheduled on a different processor than the one it last ran on must pull several data structures to the cache of its new host processor before starting to run. This explains the large amount of communication measured during this transition, which amounts to 45% (coherence plus upgrade time) of the execution time for `END_IDLE`.

The frequent rescheduling of processes on different processors increases the coherence traffic. Three data structures closely associated with processes (the process table, user areas, and kernel stacks), are responsible for 33% of the kernel's coherence misses. To hide part of the coherence miss latency, the operating system could prefetch all or part of these data structures when a process is rescheduled on another processor. The operating system may also benefit by using affinity scheduling to limit the movement of processes between processors.

Engineering workload. Kernel activity in the engineering workload is not heavily affected by the transition to multiprocessors. The `UTLB` miss handler dominates the minimal kernel time of the engineering workload. The multiprocessor `UTLB` handler contains two extra instructions, resulting in a small impact on its performance.

6 Related Work

A number of recent papers have characterized multiprogrammed and multiprocessor operating system behavior. One interesting point of comparison between these studies and ours is the methodology used to observe system behavior. Previous studies were based on the analysis of traces either using hardware monitors [2][8][13] or through software instrumentation [3]. To these traditional methodologies, we add the use of complete machine simulation for operating system characterization. We believe that our approach has several advantages over the previous techniques.

First, SimOS has an inherent advantage over trace-based simulation since it can accurately model the effects of hardware changes on the system. The interaction of an operating system with the interrupt timer and other devices makes its execution timing sensitive. Changes in hardware configurations impact the timing of the workload and result in a different execution path. However, these changes are not captured by trace-based simulations as they are limited to the ordering of events recorded when the trace was generated.

When compared to studies that use hardware trace generation to capture operating system events of interest, SimOS provides better visibility into the system being studied. For example, operating system studies using the DASH bus monitor [2][13] observe only level-2 cache misses and hence are blind to performance effects due to the level-1 caches, write buffer, and processor pipeline. Furthermore, because the caches filter memory references seen by the monitor, only a limited set of cache configurations can be examined. SimOS simulates all of the hardware and no events are hidden from the simulator. SimOS can model any hardware platform that would successfully execute the workloads.

Studies often use software instrumentation to annotate a workload's code and to improve the visibility of hardware monitors. Unfortunately, this instrumentation is intrusive. For example, Chen [3] had to deal with both time and memory dilations. Although this was feasible for a uniprocessor memory system behavior study, it becomes significantly more difficult on multiprocessors. The SimOS annotation mechanism allows non-intrusive system observation at levels of detail not previously obtainable.

Our results confirm numerous studies [2][3][8][13]: memory system performance, block copy, and instruction miss stall time are important components of operating system performance. Like Maynard [8] and Torrellas [13], who used hardware-based traces, we were able to examine large applications and confirm their results. Using SimOS, however, we were able to examine the operating system in more detail and explore the effects of technology trends.

7 Concluding Remarks

We have examined the impact of architectural trends on operating system performance. These trends include transition from simple single-instruction-issue processors to multiple-instruction-issue

dynamically-scheduled processors, moves towards higher clock-rates and larger non-blocking caches, and a transition from uniprocessors to multiprocessors. The workloads studied include program development, commercial database, and engineering compute-server environments.

Our data show that the I/O subsystem is the primary bottleneck for the program development and the database workloads, and that its importance continues to increase over time. For the program-development workload this result emphasizes the need for the removal of synchronous writes in the handling of file system meta-data. Since the database effectively bypasses the operating system by using "raw" disk devices, there is little the operating system can do about I/O problems in this case.

The memory system is the second major bottleneck for all of the workloads. While in the kernel, the processor is stalled for more than 50% of the time due to cache misses. Fortunately, architectural trends appear to be improving the situation; we find the memory stall time actually reduces slightly when moving from the 1994 to 1998 CPU model, even though the peak processor-execution rates grow very rapidly. The reasons for this are two-fold. First, the larger caches in subsequent years help reduce the miss rate, and second, the ability of dynamically-scheduled processors to overlap outstanding misses with computation helps hide the latency of misses.

Kernel builders wishing to improve performance beyond that provided by the architectural improvements should invest in techniques to improve cache reuse and to exploit the increased concurrency to be found in future memory systems. Suggested changes include having the virtual memory page allocator factor in the probability of the page being in the cache when doing allocation and re-writing all memory copy routines to optimally exploit the non-blocking caches.

The multiprocessor results show that each of the kernel services takes substantially more computation time than on the uniprocessors. The reasons vary all over for the different kernel services; they could be any combination of overhead and contention due to locks, stalls due to coherence misses, and extra instructions and/or misses due to use of different data structures. These need special attention from kernel developers since improvements in hardware are not expected to help much (e.g., larger caches do not reduce coherence misses), and changes in data structures and locking will be required.

The multiprocessor results also point out the importance of simulation environments such as SimOS in guiding future improvements in the kernel. SimOS provides detailed information regarding the cause(s) of performance bottlenecks for individual operating system services. Without such detailed understanding of the bottlenecks, especially on complex architectures as expected in the future, tremendous effort could be wasted in ad hoc optimizations that increase kernel complexity without improving performance.

Acknowledgments

We thank the conference referees, our shepherd Hank Levy, and members of the Hive team for their comments on drafts of this paper. Jim Bennett developed and provided us with the MXS simulator. We gratefully acknowledge SGI for providing us access to the IRIX source code. We thank Sybase for providing us with their system and Ben Verghese for showing us how to use it. This study is part of the Stanford FLASH project, funded by ARPA Grant DABT63-94-C-0054. Mendel Rosenblum is partially supported by a National Science Foundation Young Investigator award. Ed Bugnion and Steve Herrod are NSF Graduate Fellows. Anoop Gupta is partially supported by a National Science Foundation Presidential Young Investigator award. Emmett Witchel and much of the SimOS environment development has been supported by NSF CCR-9257104-03.

Bibliography

[1] James Bennett and Mike Flynn, "Performance Factors for Superscalar Processors", Technical report CSL-TR-95-661, Computer Systems Laboratory, Stanford University, Feb. 1995.

[2] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta, "Memory System Performance of UNIX on CC-NUMA Multiprocessors", In *Proceedings of SIGMETRICS/ PERFORMANCE '95*, pp. 1-13, May 1995.

[3] J. Bradley Chen and Brian N. Bershad, "The Impact of Operating System Structure on Memory System Performance", In *Proceedings of the 16th International Symposium on Operating System Principles*, pp. 120-133, Dec. 1993.

[4] Jim Gray, Ed., "The Benchmark Handbook for Database and Transaction Processing systems", Morgan Kaufmann Publishers, 1993.

[5] Mark Horowitz, Stanford University, Personal Communication, Aug. 1995.

[6] David Kotz, Song Bac Toh, and Sriram Radhakrishnan, "A Detailed Simulation of the HP 97560 Disk Drive", Technical Report PCS-TR94-20, Dartmouth College, 1994.

[7] Jeff Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy, "The Stanford FLASH multiprocessor", In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 302-313, April 1994.

[8] Ann M. G. Maynard, Collette M. Donnelly, and Bret R. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads", In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 145-156, Oct. 1994.

[9] MIPS Technologies, Inc., "R10000 Microprocessor User's Manual - 2nd edition", June 1995.

[10] John Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?", In *Proceedings of the Summer 1990 USENIX Conference*, pp. 247-256, June 1990.

[11] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta, "Fast and Accurate Multiprocessor Simulation: The SimOS Approach", In *IEEE Parallel and Distributed Technology*, Volume 3, Number 4, Fall 1995.

[12] Mendel Rosenblum and John Ousterhout, "The Design and Implementation of a Log-structured File System", *ACM Transactions on Computer Systems, 10(1):25-52*, Feb. 1992

[13] Josep Torrellas, Anoop Gupta, and John L. Hennessy, "Characterizing the Cache Performance and Synchronization Behavior of a Multiprocessor Operating System", In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 162-174, Oct. 1992.

[14] Josep Torrellas, Chun Xia, and Russel Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads", In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, pp. 360-369, Jan. 1995.