DECEMBER 1990

WRL
Technical Note TN-16

The Effect of Context Switches on Cache Performance

Jeffrey C. Mogul and Anita Borg

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

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

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

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

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

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

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

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

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

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

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

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

The Effect of Context Switches on Cache Performance

Jeffrey C. Mogul and Anita Borg

December, 1990

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

The sustained performance of fast processors is critically dependent on cache performance. Cache performance in turn depends on locality of reference. When an operating system switches contexts, the assumption of locality may be violated because the instructions and data of the newly-scheduled process may no longer be in the cache(s). Context-switching thus has a cost above that associated with that of the operations performed by the kernel.

We fed address traces of the processes running on a multi-tasking operating system through a cache simulator, to compute accurate cache-hit rates over short intervals. By marking the output of such a simulation whenever a context switch occurs, and then aggregating the post-context-switch results of a large number of context switches, it is possible to estimate the cache performance reduction caused by a switch. Depending on cache parameters the net cost of a context switch appears to be in the thousands of cycles, or tens to hundreds of microseconds.

This technical note is a preprint of a paper to appear in the
Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.

1. Introduction

The sustained performance of fast processors is critically dependent on cache performance. Although in principle a well-designed CPU should be able to execute nearly one instruction per cycle (assuming that it issues one instruction at a time), in practice it is not possible to sustain such an execution rate because the memory system cannot always deliver instructions and data fast enough.

Cache performance in turn depends on locality of reference; when the sequence of addresses referenced by software cannot all be stored in the cache, cache misses result. In modern computers, the penalty for a single cache miss might be tens or hundreds of cycles [11]. It is not possible to build a cache that is large enough to hold the working sets of all possible software, nor is it possible to code all software to avoid all cache misses. It is possible, however, to design caches that provide high performance for a wide range of software, and it is also possible to structure software to use the cache (or caches) efficiently [12]. The trick is thus to match the design of the cache system with the design of the software, so that the overall system will have maximum performance.

When an operating system managing multiple processes switches contexts, the assumption of locality may be violated because the instructions and data of the newly-scheduled process may no longer be in the cache or caches. This may be because the caches are simply too small to hold the working set of many processes. Context-switching thus has a cost above that associated with the operations performed by the kernel.

It is important to know what the costs of a context switch are, because if the cache-performance cost is low, or if context switching can be avoided by restructuring the operating system, then the cache system should be designed to yield the best price/performance ratio for uninterrupted application programs. On the other hand, if the cache-performance cost is high, and if context switches are expected to be frequent, then one should pay attention to this when designing the cache system. In particular, real-time systems, in which processes must respond quickly to external events, might inevitably exhibit poor locality of reference and yet require maximum CPU performance immediately after a context switch. (Designers of real-time systems typically have avoided caches, to ensure predictability. This may no longer be feasible, particularly for such applications as multi-media workstations.)

The cache-performance cost of a context switch may be estimated by looking at how the cache-hit rate varies after a context switch. In principle, this could be done by using a hardware or microcode-based monitor [1], but in practice it might be impossible for the monitor to know when a context switch has occurred, and to keep track of which process was running after the switch. It is much easier to take address traces from an actual system, annotated with context-switch information, and feed them to a cache simulator. Also, the simulation approach makes it possible to examine the effect of many different cache designs.

The experiments described in this paper measure the cache-performance cost of context switches for a variety of programs executing concurrently (through timesharing) on a Unix® operating system. Several different cache organizations were simulated, representing both current and future workstation hardware. (Since ‘‘server’’ systems are often just workstation processors in larger packages, their cache systems are the same and our results should apply.)

Depending on the cache system parameters, we found cache-performance costs up to tens or hundreds of microseconds for an average context switch. Viewed another way, a context switch ‘‘wastes’’ several thousand instruction cycles, and is comparable to the time it takes to send or receive a network packet. The cache-performance costs of a context switch may be greater than all other context-switch costs.

1.1. Previous work

Several trace-based studies have analyzed cache performance under multiprogramming workloads. When true multiprogramming traces have not been available, as in Smith [16] and in Thiebaut and Stone [18], single process traces have been interleaved based on estimates of instruction execution time. Agarwal et al. [1, 2] and Stunkel and Fuchs [17] have captured accurate multiprogramming traces; simulations using their multiprogramming traces have focused on long-term average performance for different cache configurations. For example, Agarwal used multiprogramming traces to compare the average behavior of uniprocess caches and multiprocess caches with either cache flushing on context switch or the use of process identifiers on cache lines. Hill and Smith [10] used Agarwals’s traces to evaluate associativity of caches, while Stunkel’s primary concern was multiprocessor cache behavior.

Multiprogramming traces are used because they better represent the actual sequence of memory references on a machine. To our knowledge, they have never been used to isolate and quantify the fine-grain behavior of processes after a context switch or to differentiate cache behaviors for voluntary and involuntary context switches. (Clark and Emer [7] used non-multiprogramming traces to examine how both voluntary and involuntary context switching affected translation-buffer performance.)

The short length of previous traces has also limited their usefulness. None of the earlier studies collected traces of more than tens of millions of references. Our much longer traces, of hundreds of millions or billions of references, allow us to average the short-term behavior of thousands of context switches. They also allow accurate analysis of the effects of context switches in systems with very large caches.

2. Goals and caveats

The goal of our experiments was to measure the decrease in overall system performance resulting from a context switch. Since the results of measuring any short single sequence of instructions is uninformative (because over short time scales, locality can vary significantly), we instead looked at statistics (primarily the mean) of a large number of context switches.

We were interested in the performance of the system, not in the underlying cache hit rates, because performance is a function not only of cache hit rate but also of memory system latencies. For this reason, our measurements are cast in terms of the ‘‘cycles per instruction’’ (CPI) observed when executing programs. For the single-issue architectures that we simulated, a CPI of 1.0 is ideal; higher CPIs represent poorer performance. For example, if the average CPI over the execution of a program is 2.0, then the program will take twice as long to complete as on a idealized machine that never has a cache miss.

The programs that were traced were compiled for a straightforward ‘‘RISC’’ architecture [13]. Our results are most directly applicable to systems with similar instruction sets, including many of those currently on the market. Architectures with unusually large (or small) register sets might behave differently.

Our cache simulations assume that cache misses are the only possible cause for an instruction taking more than one cycle. In real systems this may not be true; for example, pipeline interlocks may cause the processor to stall at times. Also, we include in our count of ‘‘instructions’’ executed any no-op instructions inserted by the compiler (for example, in branch slots). Because of this, the simulated CPI might be somewhat different from the real CPI, especially on machines with different instruction sets. We assume that good compilers will do enough code-reorganization to approach the optimal CPI.

We have not measured the cost of executing the kernel code for performing a context switch. Other studies have measured this cost on other systems [3, 4, 14]. Context-switching times measured with minimal user-process code (hence minimizing the cache effects covered in this paper) show large variations between operating systems, and the operating system kernel used for our experiments is known to have poor performance. For a given operating system, context-switch costs vary between different processors even after normalizing for nominal processor speed [3, 14], partly because the cost of executing the kernel code to do context switches is affected by cache performance. Because of this, kernel overhead for context-switching may increase (in relative terms) as cache-miss penalties grow with increasing cycle time. Careful measurement of this effect is a good area for further research.

Because our measurements are not predicated on any specific operating system, they should be applicable to any similar system. By ‘‘similar’’ we mean a system that reschedules processes in the same way that Unix systems do, and that does not make any attempt to take cache behavior into account when deciding which process to run next. A scheduler that tried to schedule processes to avoid context-switch-related cache misses might not produce the same results.

We also ignored any kernel operations that might flush or invalidate regions of the cache. Mostly, these would be associated with network or file input operations. This means that the actual cache performance following certain system calls might be worse than our simulations show.

3. Measurement method

Each in our series of experiments consisted of these steps:

1. Run a set of application programs concurrently on a timesharing system.
2. Trace the instruction and data addresses generated during the execution of these programs.
3. Use the trace to simulate the cache behavior for a specific cache design.
4. Extract from the simulation results the system’s performance (in terms of CPI) for some number of instructions following each context switch.

5. Aggregate the post-context-switch performance over all the context switches to compute the average cost.

The first four steps are actually performed concurrently: as the programs run and generate traces, the traces are fed ‘‘on-the-fly’’ to a cache simulator. Otherwise, the volume of the traces would have overwhelmed our disk storage. Even so, the output of the cache simulator is quite bulky, on the order of 10 Mbytes for one experiment.

We collected the results of 5000 context switches for each experiment; this reflects between 33 million and 1.3 billion user instructions executed. Several hours are required to generate each such trace, mostly due to the cost of cache simulation.

Sections 3.1 and 3.2 sketch how the traces are generated and the cache simulation is done; a more detailed description is available in [5].

3.1. Generation of traces

Address traces are generated by the programs themselves. A special linker modifies the code generated by the compiler to insert calls to tracing routines at each load and store, and at the entry to each basic block. The tracing routines simply add a record to a *trace buffer*, a large region of memory mapped in the program’s address space and managed by the operating system. Since the trace-buffer records for basic blocks show how many instructions are executed in the block, the trace buffer describes the complete set of addresses referenced by the program (ignoring some minor ambiguities about the ordering of instruction and load/store references within basic blocks).

Programs traced in this way are much larger than normal, so the records for basic blocks are made to reflect the instruction addresses as they occur in the unmodified program.

Also, traced programs take somewhat longer to execute than normal, so the operating system has been modified to reschedule processes less often than usual. The intent is to avoid causing more involuntary context switches simply because the programs are running slowly.

3.2. Cache simulation

Whenever the trace buffer becomes full, the operating system arranges for a special trace-analysis process to run. (This process is not itself traced, and while it is running the traced processes are not scheduled.) The trace-analysis program runs through the trace buffer and feeds the new traces to a cache simulation; when the trace buffer is empty, the analysis program blocks and allows the traced programs to continue.

The kernel inserts into the trace buffer additional records reflecting context switches (except for those caused by the tracing procedure). These identify which process is running at any given time, which is necessary for the cache simulator to convert the virtual addresses in the trace buffer to the physical addresses used by some kinds of caches. The context switch records include the reason why the process was suspended; this allows us to distinguish between page faults, involuntary suspensions, and each Unix system call type.

The analysis program can simulate a variety of cache designs, including 1- or 2- level caches, split or integrated instruction and data caches, write-through or write-back policies, and various values for line size, cache size, and cost of cache hits and misses. All costs are measured in terms of processor cycles, so the processor cycle time is not directly relevant.

For our purposes, we wanted to know the average CPI over relatively short intervals (1000 or 10000 instructions). We arranged for the cache simulator to output the CPI value for the first 100 intervals after every context switch; the simulator also indicates which process was running after the switch. (The processes are identified by short tags that are recycled fairly quickly; thus, there is some ambiguity that arises for processes that are created or destroyed during the experiment.) By printing CPI values for only the initial intervals after a context switch, we elide much of the enormous output from the simulator.

Because we want to know how the CPI would have behaved if the processes did not interfere with each other in the cache, we also simultaneously simulate ‘‘per-process’’ caches[1]. That is, the same reference stream is fed through two simulations, one that is perturbed by context switches, and one that is not. It is as if we ran each process on its own computer.

3.3. Aggregation over multiple context switches

The output of the simulator is thus a file that (in effect) contains tuples of the form:

(process-id, context-switch-ID, instructions-since-context-switch, CPI_{shared}*, $CPI_{private}$)*

where *context-switch-ID* identifies the most recent context switch event, CPI_{shared} is the CPI simulated for a real machine with one set of caches, and $CPI_{private}$ is the CPI simulated for a hypothetical machine with a separate set of hardware caches for each process. (The CPI values are the average CPI since the previous tuple.)

The first phase of aggregation is to superimpose these tuples so that for each process-id, we have a sequence of tuples of the form:

(instructions-since-context-switch, average-CPI_{shared}*, average-*$CPI_{private}$*)*

(Actually, we maintain additional information in these tuples, allowing us to calculate other statistics such as standard deviation and maximum CPI).

From this information, we can directly produce graphs of instantaneous CPI vs. instructions-since-context-switch for each of the processes. Because different programs behave in different ways (and because some of the processes do not run for long enough to generate stable results), it is more useful to aggregate the per-process results into overall statistics for the ‘‘average’’ context switch (for example, see figure 2).

Plots of CPI vs. instructions-since-context-switch give a good sense of how long, in terms of instructions executed, it takes after a context switch for the cache performance to return to normal. To get a sense of how long it takes in terms of actual time (or at least, processor cycles), we

[1]More precisely, we simulate ‘‘per-process-ID’’ caches. This will introduce some small errors when process IDs are reused, but this happens infrequently.

calculate the ‘‘excess CPI’’ as the difference between the CPI_{shared} during a specific interval and the $CPI_{private}$ for the same interval (e.g., figure 4). We can then integrate this curve with respect to time to get the ‘‘cumulative excess cycles,’’ which is effectively the cache-performance cost of a context switch (e.g., figure 5).

4. Experiments

Since some application mixes cause more context switches than others, we chose three sets of programs. One simulates a timesharing system with a few intensive users; another simulates a compute-bound load with a couple of large programs; the third simulates a repetitive client-server interaction, such as might be encountered using a kernelized operating system. The programs are summarized in table 1; most of the programs each create several processes. The benchmark sets are summarized in table 2; the sizes listed are approximate.

PROGRAM	DESCRIPTION	MAXIMUM SIZE
Magic	A VLSI layout system.	10 Mb
Tree	A memory-intensive tree-search program written in Scheme.	12 Mb
BigTree	*Tree* on a larger problem.	65 Mb
TV	A CMOS VLSI timing verifier.	3.4 Mb
make #1	Compilation of portions of the C source for *Magic*.	
make #2	Link and load of the *Magic* program binaries.	
interactive	An infinitely looping shell script of simple commands (*cp, cat, ex, rm, ps,* and *ls*).	
pingpong	A pair of processes exchanging messages, as if one were a file server and the other a client.	0.6 Mb

Table 1: Programs used to generate traces

NAME	PROGRAMS	TOTAL SIZE
Timesharing	*Magic, Tree, make #1, make #2, interactive*	75 Mb
Compute-bound	*BigTree, TV*	69 Mb
Client-Server	*pingpong*	0.6 Mb

Table 2: Benchmark sets

4.1. Cache configurations and parameters

We simulated a variety of different cache designs, which are summarized in table 3. All of the designs are direct-mapped caches. Cache tags are either physical addresses or virtual addresses with process-ID tags, so the caches need not be flushed on context switches. The word size in all cases is 32 bits.

NAME	**LEVEL 1 INSTRUCTION**	**LEVEL 1 DATA**	**LEVEL 1 MISS TIMING**	**LEVEL 2 UNIFIED**	**LEVEL 2 MISS TIMING**
Design 1a	16B x 4k = 64kB	16B x 4k = 64kB Write through 6 write buffers	15 cycles	None	
Design 1b	16B x 16k = 256kB	16B x 16k = 256kB Write through 6 write buffers	15 cycles	None	
Design 2a	16B x 128 = 2kB	16B x 128 = 2kB Write through 4 write buffers	13 cycles	128 x 4k = 512kB Write back	200 cycles
Design 2b	16B x 512 = 8kB	16B x 512 = 8kB Write through 4 write buffers	13 cycles	128 x 16k = 2mB Write back	200 cycles

Sizes are in bytes   Level-1 hit timing = 1 cycle, in all cases   Level-2 hit timing = Level-1 miss timing

Table 3: Cache configurations

Design 1a represents the memory system of a contemporary high-performance workstation, the DECStation™-5000/200, with a cycle time of 40 nSec[2]. Design 1b represents the same system, except that the cache RAMs are one generation larger (RAM sizes usually grow by a factor of 4 between generations).

Design 2a represents a hypothetical future workstation design based on a single-chip processor with on-chip level 1 caches, and an external level-2 cache. The cycle time for such a CPU might be between 2 nSec and 4 nSec. Design 2b represents the same system as used in Design 2a, with cache RAMs four times as large.

5. Results

For each set of benchmarks, we ran trials using each of the cache designs. In section 5.4, we summarize the cache-performance costs of context switching for each of the trials. Before that, however, we will examine a small set of trials in detail, in order to see how we obtained the summary results. These particular trials were all done using cache design 1a.

5.1. Timesharing benchmark results (Trial A)

We begin by looking at what kinds of context switches were captured in our traces. In a trace, each context switch is followed by CPI values for some number of instructions, before another context switch occurs; we will call each of these series of CPI values a ‘‘trace segment.’’ Figure 1 graphs the distribution of trace segment lengths. (The point where the solid curve passes the 50% level on the *y*-axis gives the median interval between context switches.) The data are also broken down by the causes of the switches: system calls, page faults, and involuntary switches. Even though the *Timesharing* workload involves a mix of IO-bound and compute-bound

[2]The actual cycle time has no effect on our simulations, since all times are normalized to units of cycles.

programs, most of the context switches are involuntary, imposed by the scheduler[3].

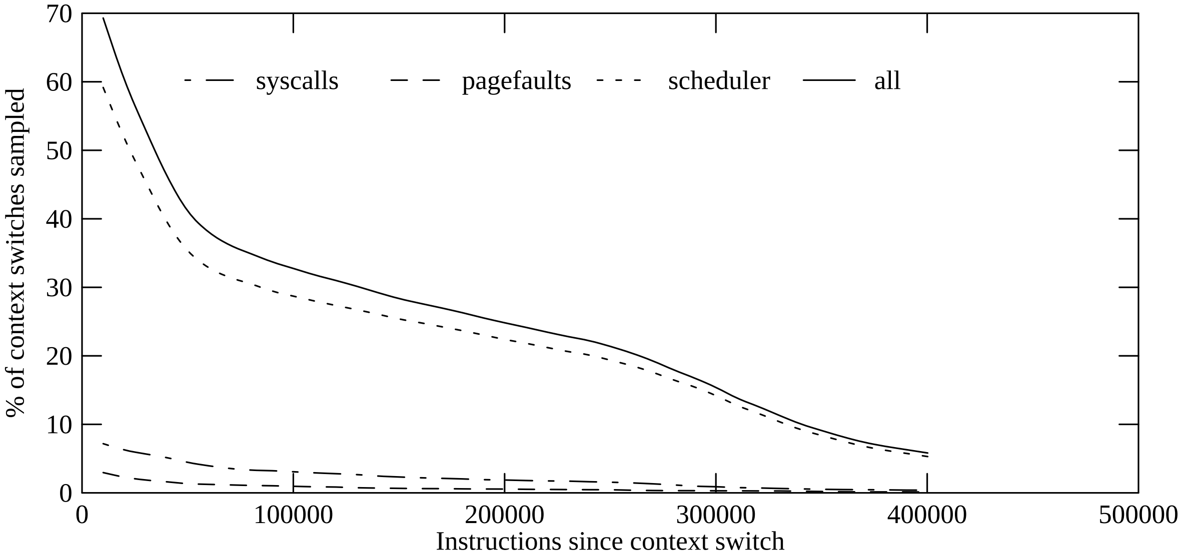


Figure 1: Distribution of trace segment lengths (Trial A)

Next we graph CPI as a function of instructions since context switch. Figure 2 shows CPI values obtained when simulating a shared cache; figure 3 shows the values obtained when simulating one cache per process. The curves are averages over all the context switches in the trace. Again, the data has been broken down into three classes of context-switch causes.

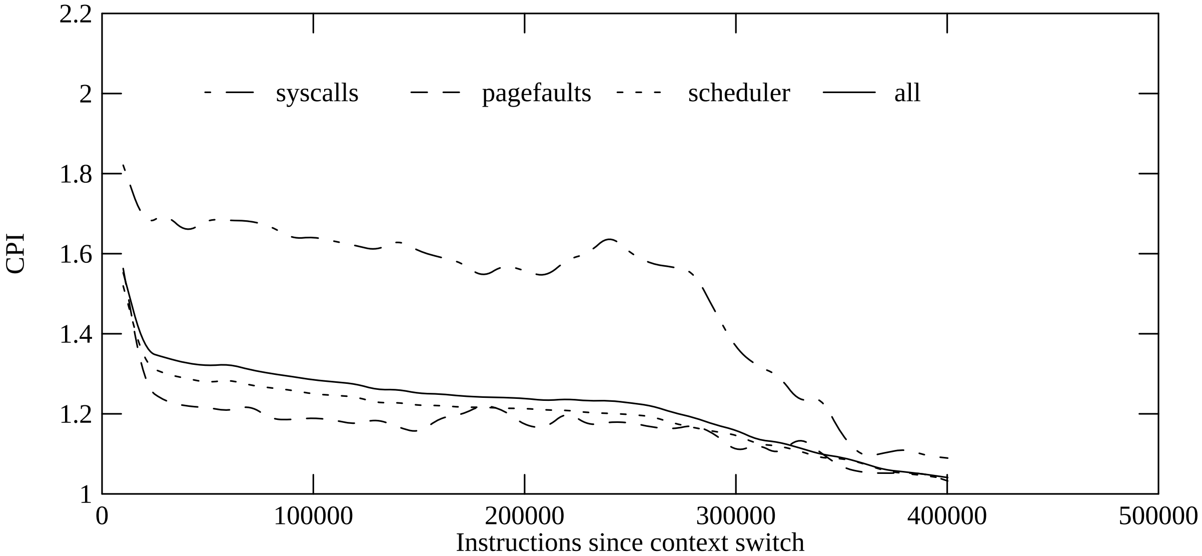


Figure 2: Average CPI for all switches, shared cache (Trial A)

[3]Since traced programs run slower than normal, the scheduler has been modified to let them for about 400,000 cycles, or 1/60th of a second of non-traced time.

Looking at figure 3, we see that context switches affect cache performance even when multiprogramming is ignored. System calls have the greatest effect on CPI, because after a system call (a file read, for example) a process is likely to be operating on new data. A page fault also increases the CPI, because it signals that the process has switched to a new locality of reference.

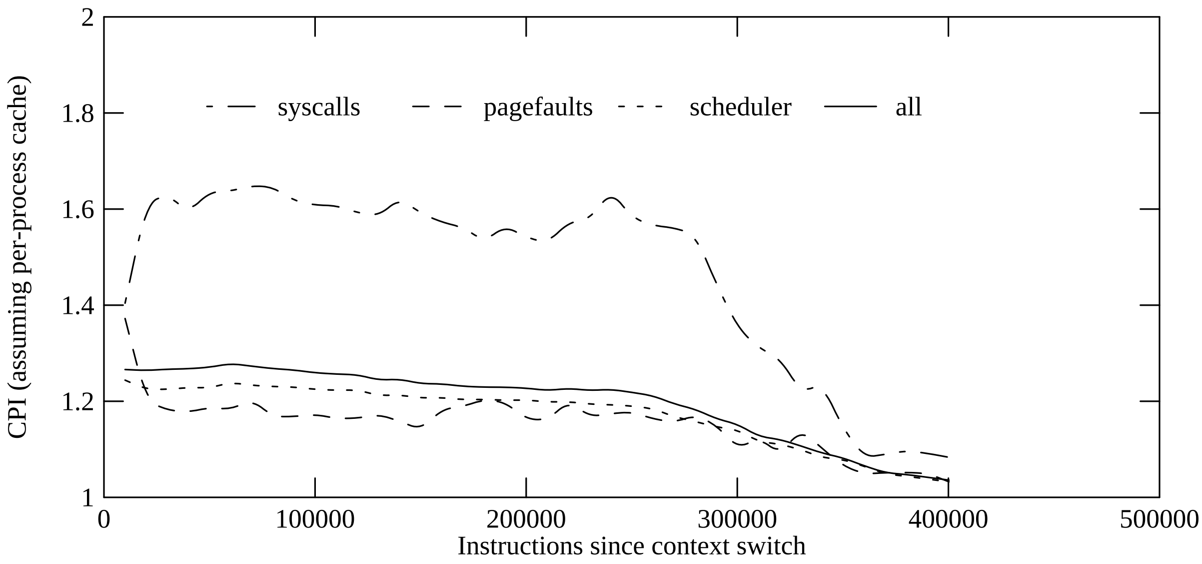


Figure 3: Average CPI for all switches, private caches (Trial A)

Involuntary switches, on the other hand, should not affect the non-multiprogramming CPI of a process, since they are not correlated with the process's addressing behavior. The ''scheduler'' curve in figure 3 is certainly flatter than the others, yet it is not completely flat.

Why is this so? It is because the curve is an aggregate over all the processes in the benchmark, and all the involuntary context switches. Although the curve runs the width of the graph, not all of the individual trace segments last that long. For a compute-bound process, which seldom switches voluntarily, its segments show steady-state behavior and a relatively low CPI. I/O-intensive processes, however, voluntarily switch so often that they seldom reach steady-state behavior. When involuntarily suspended, their CPI is still relatively high, and when next scheduled they usually switch voluntarily after a short while. Thus, segments from these processes are shorter, and contribute their high CPI only to the first part of the aggregated curve[4].

The curve in figure 3 for switches caused by system calls also shows surprising behavior: it has a notable initial dip (directly visible only when graphed at a finer time-scale). The full explanation of this is straightforward but too lengthy to fit here. Briefly, it comes because virtually all of the system-call context switches in the benchmark come from a single loop in one of the processes, and this process apparently does something unusual. Because few of the context switches are caused by system calls (see figure 1), this anomaly has little effect on the overall average.

---

[4]Another way to think about this curve is as the aggregate of a number of nearly-flat lines of varying heights and lengths.

Figure 4 shows the difference between figures 2 and 3; that is, it shows the portion of the CPI attributable to multiprogramming. We call this the ‘‘excess CPI.’’ Note that the curve for ‘‘scheduler switches’’ is indistinguishable from the curve for all switches; this is because most switches are involuntary.

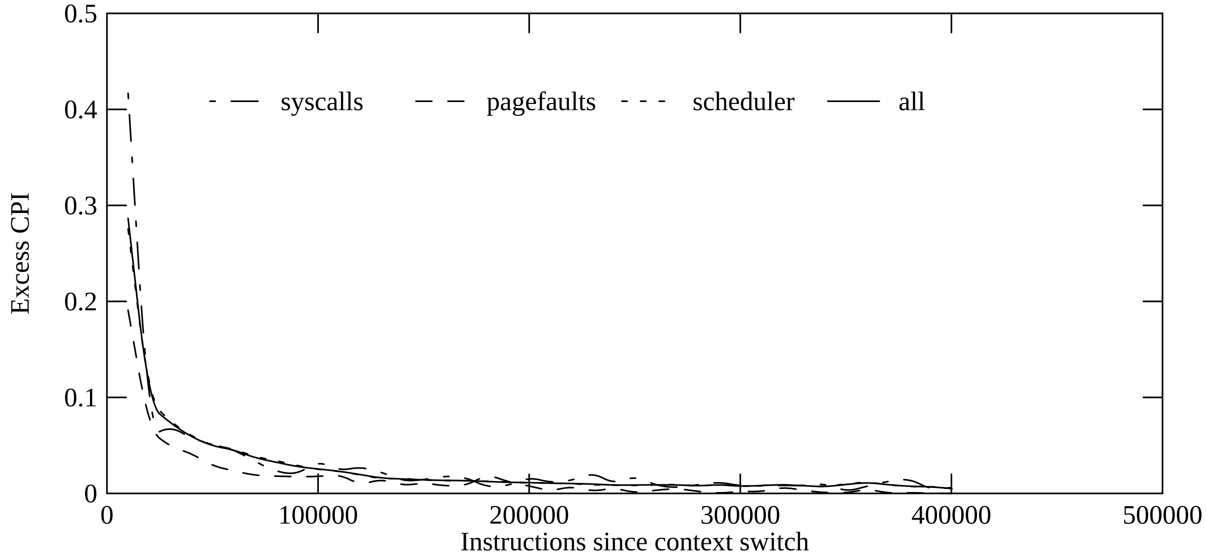


Figure 4: Excess CPI for all switches (Trial A)

Finally, we can calculate the total cache-performance cost of a context switch, by integrating the curves in figure 4 with respect to instructions since a context switch. We call this the ‘‘cumulative excess CPI’’; figure 5 shows the results for this trial.

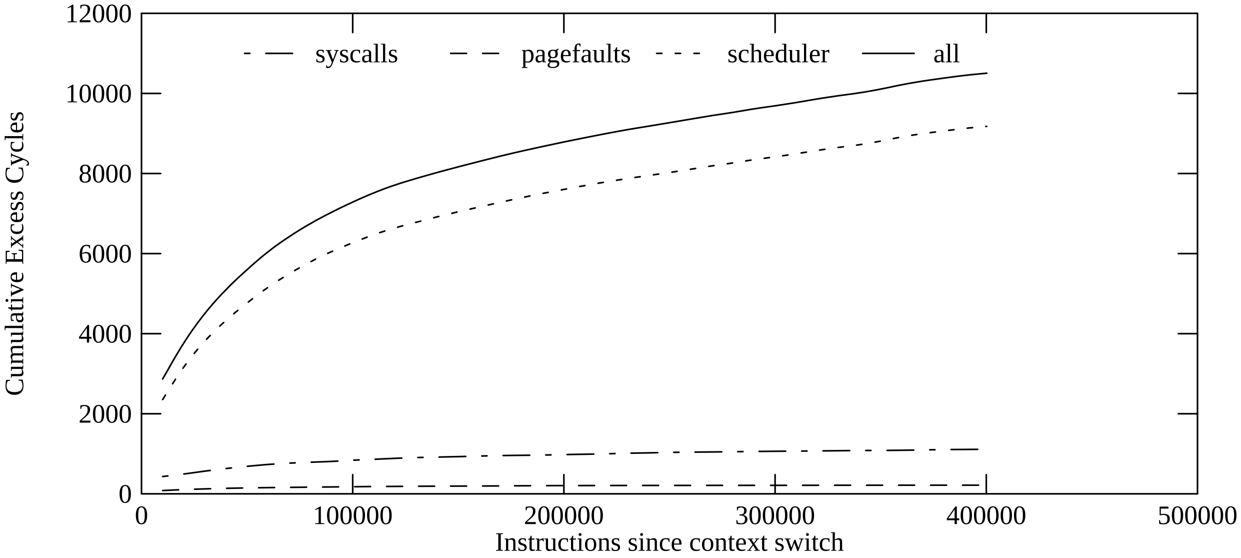


Figure 5: Cumulative excess CPI for all switches (Trial A)

One can see from figure 5 that the curve does not reach an asymptote. If we were to look further out in time, it would flatten out, but since (as we saw in figure 1) few of the trace segments last that long, the results in that region are suspect. Thus, we arbitrary cut off the curve at the point where only 10% of the trace segments remain, and call the final value the average cost for the trial. This number may not have any intrinsic meaning, but it allows us to make useful comparisons between different cache configurations and benchmarks.

It might seem surprising that the effects of a context switch are detectable, however weakly, 400,000 instructions later, but we believe that the effect is real. In the *Timesharing* benchmark, the scheduler rotates through enough processes[5] that by the time a process is rescheduled, most of its working set has been removed from the cache; in other words, a context switch might effectively cold-start the cache. The combined cache contains 8k lines, and a miss costs 15 cycles, so it could take over 120,000 cycles to reload a completely discarded working set. The average context-switch cost we calculated is an order of magnitude smaller than this, so we conclude that a process might well be subject to excess cache misses long after a context switch.

5.2. Compute-bound benchmark results (Trial B)

The *Compute-bound* workload, which consists of two compute-bound processes sharing the CPU, generates almost exclusively involuntary context switches. (Fewer than 1% are caused by either system calls or page faults.) These processes have a slightly lower average CPI than those in the *Timesharing* workload, probably because they do more computation between memory references. The excess CPIs in this benchmark are lower as well, so the resulting curve for cumulative excess CPI reaches a lower plateau (see figure 6).

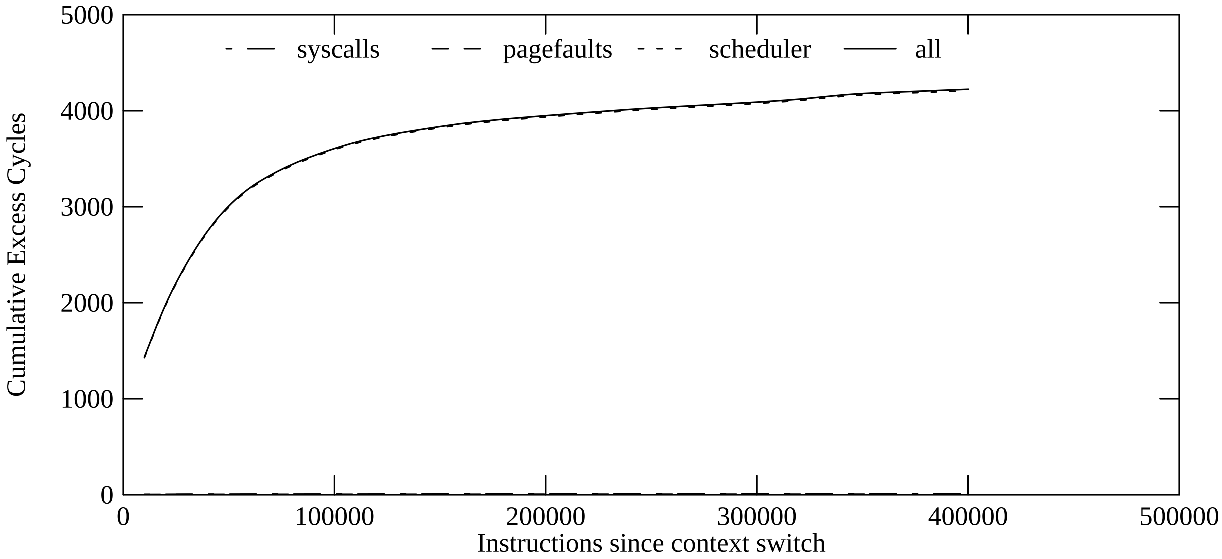


Figure 6: Cumulative excess CPI for all switches (Trial B)

[5]The number of processes varies between trials, averaging 85, but not all 85 are runable at once.

5.3. Client-Server benchmark results (Trial C)

The *Client-Server* workload, which consists of two intensively communicating processes alternating the CPU, generates almost exclusively voluntary context switches. (Fewer than 0.1% are involuntary, and no page faults are taken.) The CPIs are extremely low, since almost all the references are satisfied by the cache. In fact, the private-cache CPIs are almost exactly 1.0, but there is a little contention in the shared cache case, so the excess CPIs are non-zero. Figure 7 shows the cumulative excess CPI; the curve is flat after 4000 instructions, because from that point on, the excess CPI is zero.

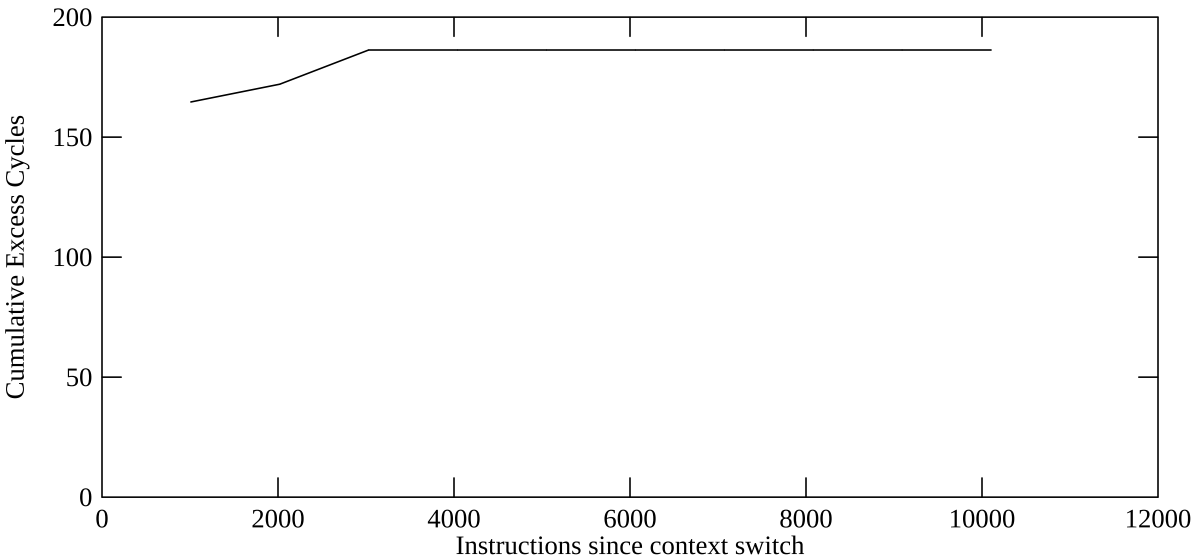


Figure 7: Cumulative excess CPI for all switches (Trial C)

5.4. Summary of all experiments

The experiments we ran are summarized in table 4. ‘‘Data refs’’ gives the number of loads and stores executed. ‘‘Average cost’’ is the average, over all the processes for the entire trace, of the cost of a context switch. ‘‘Total cycles’’ gives the total number of cycles simulated for the trace.

The column labelled ‘‘Overhead ratio’’ is an indication of the total slowdown due to the cache-performance costs of context switching. It is calculated by multiplying the ‘‘Average Cost’’ by the number of switches (5000) to get the total number of excess cycles, and dividing that by ‘‘Total cycles’’ to get a ratio.

We ran at least two trials for each combination of benchmark and cache design. Because our cache simulations were done ‘‘on-the-fly’’, each trial represents a different trace; thus, there is some variation between experiments due to luck rather than cache parameters.

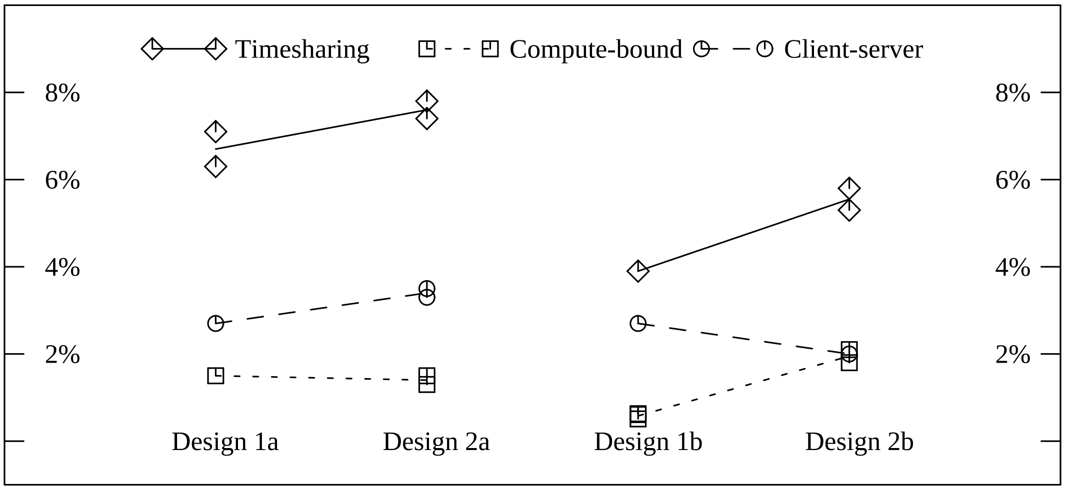
TRIAL	CACHE DESIGN	BENCHMARK SET	TOTAL INSTRS	DATA REFS	AVERAGE COST	TOTAL CYCLES	OVERHEAD RATIO
A	Design 1a	*Timesharing*	$555 * 10^6$	$149 * 10^6$	10000 cycles	$706 * 10^6$	7.1%
A1	Design 1a	*Timesharing*	$651 * 10^6$	$174 * 10^6$	10300 cycles	$820 * 10^6$	6.3%
B	Design 1a	*Compute-bound*	$1313 * 10^6$	$356 * 10^6$	4300 cycles	$1445 * 10^6$	1.5%
B1	Design 1a	*Compute-bound*	$1238 * 10^6$	$335 * 10^6$	4200 cycles	$1361 * 10^6$	1.5%
C	Design 1a	*Client-Server*	$33 * 10^6$	$10 * 10^6$	186 cycles	$35 * 10^6$	2.7%
C1	Design 1a	*Client-Server*	$33 * 10^6$	$10 * 10^6$	186 cycles	$35 * 10^6$	2.7%
D	Design 2a	*Timesharing*	$538 * 10^6$	$145 * 10^6$	24500 cycles	$1561 * 10^6$	7.8%
D1	Design 2a	*Timesharing*	$635 * 10^6$	$169 * 10^6$	26300 cycles	$1787 * 10^6$	7.4%
E	Design 2a	*Compute-bound*	$1229 * 10^6$	$333 * 10^6$	11700 cycles	$3791 * 10^6$	1.5%
E1	Design 2a	*Compute-bound*	$1235 * 10^6$	$335 * 10^6$	10000 cycles	$3800 * 10^6$	1.3%
F	Design 2a	*Client-Server*	$33 * 10^6$	$10 * 10^6$	261 cycles	$37 * 10^6$	3.5%
F1	Design 2a	*Client-Server*	$33 * 10^6$	$10 * 10^6$	234 cycles	$36 * 10^6$	3.3%
G	Design 1b	*Timesharing*	$649 * 10^6$	$175 * 10^6$	5500 cycles	$713 * 10^6$	3.9%
G1	Design 1b	*Timesharing*	$732 * 10^6$	$196 * 10^6$	6300 cycles	$804 * 10^6$	3.9%
H	Design 1b	*Compute-bound*	$1271 * 10^6$	$345 * 10^6$	1700 cycles	$1350 * 10^6$	0.63%
H1	Design 1b	*Compute-bound*	$1233 * 10^6$	$334 * 10^6$	1340 cycles	$1310 * 10^6$	0.51%
H2	Design 1b	*Compute-bound*	$1310 * 10^6$	$355 * 10^6$	1700 cycles	$1391 * 10^6$	0.61%
I	Design 1b	*Client-Server*	$33 * 10^6$	$10 * 10^6$	186 cycles	$35 * 10^6$	2.7%
I1	Design 1b	*Client-Server*	$33 * 10^6$	$10 * 10^6$	186 cycles	$35 * 10^6$	2.7%
J	Design 2b	*Timesharing*	$619 * 10^6$	$164 * 10^6$	12500 cycles	$1170 * 10^6$	5.3%
J1	Design 2b	*Timesharing*	$680 * 10^6$	$183 * 10^6$	15000 cycles	$1304 * 10^6$	5.8%
K	Design 2b	*Compute-bound*	$1201 * 10^6$	$325 * 10^6$	9200 cycles	$2191 * 10^6$	2.1%
K1	Design 2b	*Compute-bound*	$1202 * 10^6$	$325 * 10^6$	7700 cycles	$2189 * 10^6$	1.8%
L	Design 2b	*Client-Server*	$33 * 10^6$	$10 * 10^6$	143 cycles	$36 * 10^6$	2.0%
L1	Design 2b	*Client-Server*	$33 * 10^6$	$10 * 10^6$	142 cycles	$36 * 10^6$	2.0%

Table 4: Summary of experiments

5.5. Effects of changing technology

As the underlying technologies improve, processor cycle times are decreasing, and memory sizes are increasing. However, interconnection and other constraints limit the improvement in main-memory access times; the net effect is that the relative speed of main memory gets worse. Thus, cache-miss effects are also getting worse; the absolute size of the caches must increase simply to maintain a given CPI ratio.

Does this mean that the relative cost of context-switching is also getting worse? Since we have examined only two basic memory system designs, we cannot say anything conclusive, but figure 8 hints at an answer. For the *Timesharing* benchmark, the overhead ratio does worsen for the more advanced machine. The other benchmarks do not show consistent trends.



Markers show individual trials; lines connect average values.

Figure 8: Overhead ratio vs. Technology

5.6. Effects of increased cache size

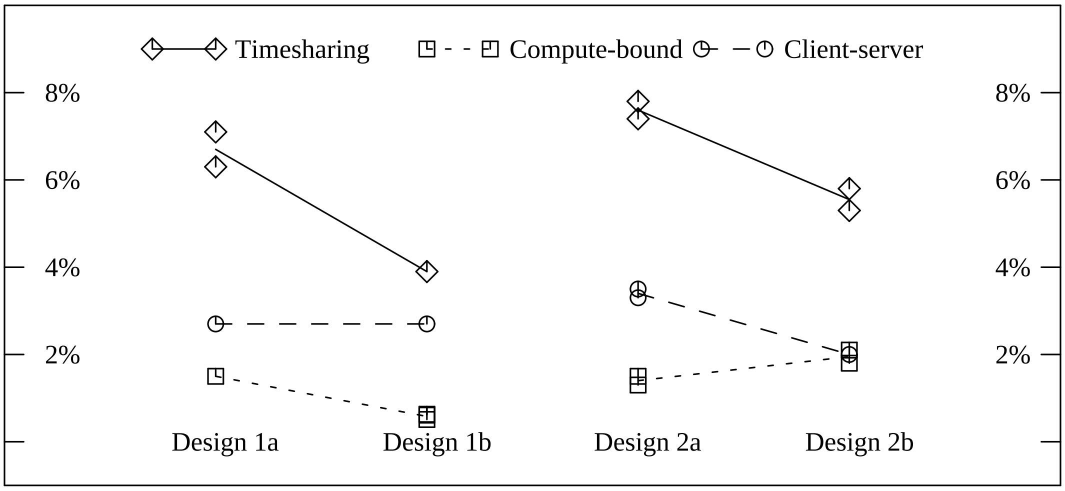
One technology effect that should improve cache performance is the continuing increase in cache RAM density. Our cache simulations allow us to examine the effect of increasing cache size by a factor of four (one generation, since RAMs are squares). Figure 9 shows how the overhead ratio varies with cache size, for the two different basic system designs.

Again, the *Timesharing* benchmark shows a consistent trend of improving overhead ratio with increased cache size. The trends for the other benchmarks are less consistent. One of the benefits of larger caches is that they preserve more data across at least some kinds of context switch; this effect is probably most important in the *Timesharing* workload.

6. Discussion

The most interesting aspect of the results of our experiments is the magnitude of the context-switch effect on cache performance. With the DECstation-5000, for example, the cost ranges from 10 microseconds (client-server benchmark) to 400 microseconds (timesharing benchmark), and our cache simulations probably underestimate the costs (by ignoring kernel references).

At the low end of this range, the cache-performance cost of a context switch is comparable to the other costs of a context switch, and at the high end it dominates the costs. For example, the Ultrix™ kernel overhead required to perform a single context switch has been measured on the DECstation-5000 at about 70 microseconds [14], in such a way as to avoid most of the cache-



Markers show individual trials; lines connect average values.

Figure 9: Overhead ratio vs. Cache size

performance effects. Anderson et al. estimate the minimum cost for doing a context switch on the DECstation-5000 at 7.4 microseconds [3].

In particular, the cost of saving and restoring even a large set of registers is quite small in comparison to the cost of refilling the cache. This should be kept in mind when designing an architecture, since large register sets may have compensating benefits [19].

A trend hinted at, although not proven, by our experiments is that as CPU cycle times continue to drop, the cache-performance costs of context switching will get worse, in relative terms. While increasing the size of the cache improves the relative costs, feasibility and economics impose constraints on cache sizes, so this trend may be unavoidable.

It may be appropriate to consider this effect when designing the memory system for a given computer. If the intended application involves a lot of context-switching, it might pay to use larger caches than would be chosen for a compute-bound system.

Operating system designers might infer from our results that performance would be improved by reducing either the cost or the frequency of context switches.

For example, it might pay to be careful about choosing the order in which to schedule processes. Also, context switching costs are relatively large when compared against the cost of exchanging packets on a local area network. A pair of minimal-length (but useful) packets can be exchanged on an Ethernet in about 130 microseconds, ignoring processing costs; on an FDDI network, the exchange can be done ten times faster. An operating system might choose to return control to a recently-suspended process when that process receives an incoming packet, on the assumption that the process's working set is still mostly in the caches.

More generally, operating system designers might want to consider these results before wholeheartedly embracing the trend towards ‘‘kernelized’’ organizations. Such systems require more context-switching than conventional ‘‘monolithic kernel’’ systems (such as Unix systems), because commonly-used system services are implemented in separate processes. (Examples include Mach [9] and V [6].) The benefits of this style of organization may outweigh the relatively small additional cache-performance costs, but if the cache effects are indeed worsening as technology progresses, decisions based on current hardware may not be applicable in the future.

6.1. Further work

The most critical deficiency of our experiments is our inability to account for the effect of kernel references on cache performance, and for the effect of context switching on kernel-reference cache performance. Our current address-tracing implementation runs on an obsolete architecture, whose operating system has excessively long code paths. While we can generate traces with kernel references, we believe that we would learn nothing beyond verifying the inefficiency of our kernel.

We are therefore in the process of porting our tracing implementation to the MIPS architecture, in a way that requires little kernel modification. This will allow us to trace benchmarks under several different, well-tuned kernels, and so to get useful information about kernel effects. It will also allow us to compare the effects of different operating system designs, since Ultrix, Mach, and Sprite [15] already run on the MIPS architecture, and other systems are being ported. It would also be interesting to measure a system whose scheduler has been experimently modified to minimize cache disruption.

We believe that it is important also to characterize larger varieties of programs and memory system designs. The port to the MIPS architecture will help, by giving us access to a wider set of programs, and by providing a large base of faster machines on which to do our experiments.

We use a crude method to simulate multiple caches using a single pass through the address trace; it requires memory proportional both to the number of processes and to the cache sizes, and so is infeasible for some experiments. More elegant methods have been described [10], and perhaps could be adapted to this problem.

It would also be interesting to examine the effects of certain novel cache designs, such as stream buffers [11], on context-switch costs. Perhaps there are less expensive ways to reduce context-switch overhead than simply increasing cache size.

One possible use of our tracing technique is as a performance-debugging tool. It allows a programmer to discover those regions of poor memory-system performance that are correlated with context switches from system calls or page faults. In many cases, an increased CPI is unavoidable, but in some instances it may indicate an inefficient algorithm. Other tools have been developed for similar purposes [8], but they are not able to show how a multi-process application might best be improved.

Alternative analyses of our address traces could answer other interesting questions, such as how much of a process’s working set is lost on each context switch, what portion of the working set is restored before the next context switch, and what are the relative contributions of the instruction and data caches.

7. Summary and Conclusions

We have developed a new approach for evaluating the cache-related performance impact of multiprogramming, and in particular the dynamic behavior of the memory system directly following a context switch. Our traces allow us to distinguish between cache misses that are intrinsic to a program, and those that are caused by collisions with other processes. We are also able to see how the different causes of context switches affect cache performance in different ways.

One implication of our results is that arguments about the proper size of register sets should be kept in perspective; the effect of register-saving on context switching may not be as important as has been thought. Another implication is that highly-modularized operating system designs might not provide the same relative performance on future hardware as they do on current hardware. Finally, when designing the memory hierarchy of a multi-programmed computer system, one should not choose the cache size based on benchmarks of single programs.

8. Acknowledgements

John Ousterhout deserves the credit for convincing us to write this paper, after we showed him some preliminary results about a week before the conference deadline. John also helped us understand the implications of our data, suggested a number of additional experiments to perform, and reviewed several drafts of the paper. Members of the ASPLOS program committee, especially Joel Emer, suggested a number of important changes to our methodology, which made the paper much stronger. David Wall helped with proofreading.

9. References

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

[2] Anant Agarwal, Mark Horowitz, and John Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems* 7(2):184-215, May, 1989.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. Palo Alto, CA, April, 1991.

[4] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. In *Proc. 12th Symposium on Operating Systems Principles*, pages 102-113. Litchfield Park, AZ, December, 1989.

[5] Anita Borg, R. E. Kessler, and David W. Wall. Generation and Analysis of Very Long Address Traces. In *Proc. 17th Annual International Symposium on Computer Architecture,* pages 270-279. Seattle, WA, May, 1990.

[6] David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter. Binary Emulation of Unix using the V Kernel. In *Proc. Summer 1990 USENIX Conference,* pages 73-85. June, 1990.

[7] 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):31-62, February, 1985.

[8] Aaron Goldberg and John Hennessy. *MTOOL: A method for detecting memory bottlenecks*. Technical Note TN-17, Digital Equipment Corporation Western Research Laboratory, December, 1990.

[9] David Golub, Randall Dean, Alessandro Forin, Richard Rashid. Unix as an Application Program. In *Proc. Summer 1990 USENIX Conference*, pages 87-95. June, 1990.

[10] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers* 38(12):1612-1630, December, 1989.

[11] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *17th Annual International Symposium on Computer Architecture*, pages 364-373. IEEE, Seattle, WA, May, 1990.

[12] Scott McFarling. Program Optimization for Instruction Caches. In *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183-191. Boston, MA, April, 1989.

[13] Michael J. K. Nielsen. *Titan System Manual*. Research Report 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

[14] John K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proc. Summer 1990 USENIX Conference*, pages 247-256. Anaheim, CA, June, 1990.

[15] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer* 21(2):23-26, February, 1988.

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

[17] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. In *Proc. ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, pages 70-78. May, 1989.

[18] Dominique Thiebaut and Harold S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems* 5(4):305-329, November, 1987.

[19] David W. Wall. Experience with a Software Architecture. To appear. 1991

WRL Research Reports

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

WRL Technical Notes

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

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

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

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

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

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Technical Note TN-15, December 1990.

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

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

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

Table of Contents

List of Figures

List of Tables