Analysis of Multithreaded Microprocessors under Multiprogramming

David E. Culler
Michial Gunter
James C. Lee

Computer Science Division — EECS
University of California, Berkeley

Report No. UCB/CSD 92/687
May 1992

Abstract: Multithreading has been proposed as a means of tolerating long memory latencies in multiprocessor systems. Fundamentally, it allows multiple concurrent subsystems (cpu, network, and memory) to be utilized simultaneously. This is advantageous on uniprocessor systems as well, since the processor is utilized while the memory system services misses.

We examine multithreading on high-performance uniprocessors as a means of achieving better cost/performance on multiple processes. Processor utilization and cache behavior are studied both analytically and through simulation of timesharing and multithreading using interleaved reference traces. Multithreading is advantageous when one has large on-chip caches (32 kilobytes), associativity of two, and a memory access cost of roughly 50 instruction times. At this point, a small number of threads (2-4) is sufficient, the thread switch need not be extraordinarily fast, and the memory system need support only one or two outstanding misses. The increase in processor real-estate to support multithreading is modest, given the size of the cache and floating-point units.

A surprising observation is that miss ratios may be lower with multithreading than with timesharing under a steady-state load. This occurs because switch-on-miss multithreading introduces unfair thread scheduling, giving more CPU cycles to processes with better cache behavior.

1 Introduction

The high-performance workstation of the mid to late 1990s is expected to comprise a microprocessor running above 100MHz, issuing multiple instructions per cycle with very fast floating-point, a substantial on-chip cache, a very large dRAM main store, and a sophisticated memory system in between. It will be used via a graphical interface with a host of windows, daemons, and background tasks running simultaneously. Extrapolating from past growth rates, a dRAM access will require the equivalent of many instruction times. This raises the interesting question of how best to focus the storage hierarchy on the processing resources. One possibility is to provide intermediate levels of caching to reduce the average off-chip access latency. However, with large on-chip caches the references that escape from the processor have relatively little spatial or temporal density. Thus, to achieve a significant hit rate, the next level cache must be very large. At this design point, the processor is expensive, the intermediate cache is expensive, the main store is expensive, and two of the three systems are idle much of the time. More careful cost analysis suggests an alternative design point. It is expected that the cache occupies half the chip or more, the floating-point units occupy roughly one quarter, and the memory management unit and the basic instruction processing unit occupy roughly one tenth each. (The Intel i860xp fits this estimate quite closely.) Replicating the register file and portions of the integer datapath represents a small investment and, given that several processes are running simultaneously, could lead to better throughput and interactivity. A process runs until it misses in the on-chip cache, at which point the miss request is issued to the off-chip memory system and the processor switches to another process (or thread) on hot stand-by. Although multithreaded processor design is non-trivial, the cost effectiveness comes from eliminating the large intermediate level cache and making concurrent use of the memory system and the processor. The processor executes instructions from one process while the memory system services a miss for a waiting process.

The goal of this paper is to identify the technology point at which the multithreaded system organization is attractive enough to warrant the engineering required to put it into practice. The question we ask is under what conditions is multithreaded execution of several concurrent processes markedly superior to timesharing amongst the same concurrent processes. This is quite different from the question of multithreading vs. single threaded execution of one process, because in our comparison both alternatives experience significant cache interference. The key issue is the impact on memory system performance, so we need to compare timesharing and multithreading across a range of cache sizes, associativities and memory latencies for various numbers of threads.

Studies of multithreaded architectures have primarily focused on multiprocessor systems where several threads of a parallel program are maintained on each processor[1, 2, 3, 4, 5, 8] In that context, it makes sense to consider switching on every remote load or on every instruction, as in [9]. We consider switching on each miss to the on-chip cache. If only one thread is used, the machine behaves as a conventional processor. Adding threads provides very fine grain timesharing. Notice that tolerating latency on multiprocessors results in concurrent use of three resources: processor, network, and memory system. The observed latency is the time a process spends in the latter two subsystems. The situation is similar with only one processor, but the network is very fast. In the uniprocessor case, however, we expect the threads to be heterogeneous and independent.

In Section 2, a simple analytical model of multithreaded execution is presented to provide a general understanding of how such systems behave. The model focuses on miss rate, which determines thread length. Section 3 outlines the empirical method used to compare memory system performance and processor utilization under timesharing and multithreading. The basic approach is to multiplex traces from several independent threads. However, an important issue arises due to the staggered completions of the traces. Section 4 presents data on the miss rates across a wide range of system configurations and identifies a minimum viable cache configuration for multithreading. Section 5 demonstrates the unfairness of thread scheduling under switch-on-miss multithreading and examines its effects on cache miss rate. Section 6 examines the processor utilization of a viable cache configuration against memory latency, switch cost, and the number of threads supported within the processor. The data indicate a potential increase in throughput of roughly 50% due to multithreading with a modest increase in processor complexity.

2 Analytical Model

To understand how a multithreaded processor should behave, we consider first a simple analytical model[6]. A thread executes on the processor for a run of R instructions.[1] It issues a memory request which takes L cycles to complete. In the meantime, the processor switches to a new thread after a switch delay of C cycles, assuming one is ready. If no thread is ready, the processor idles.

[1]We will treat the average instruction time as our basic time unit and generally refer to it as a cycle. For superscalar designs this may be smaller than the clock cycle time, but since the processor is fixed across the comparison the actual units do not matter.

The utilization is

$$\epsilon = \frac{Busy}{Busy + Switching + Idle},$$

where $Busy$, $Switching$, and $Idle$ are the amounts of time the processor spends in the corresponding states.

Let N represent the number of threads supported by the processor. If N is sufficiently large, then the L cycles required to service a given memory request will elapse before all the other ready threads have had a turn at the processor. Thus, the memory latency is entirely masked and the utilization of the processor is determined by run length and switch delay. In this case, we say the processor is *saturated*, since increasing the number of threads has no effect on performance. The processor utilization in saturation is given by the following.

$$\epsilon_{sat} = \frac{R}{R + C}$$

If the run length, switch delay, and latency are fixed, the saturation point is $N_{sat} = \frac{L}{R+C} + 1$.

On the other hand, if the latency is very large and the number of threads small, then the useful work of all the threads will be exhausted before a memory request completes. Hence, the utilization increases linearly with the number of threads, and is given by the following.

$$\epsilon_{lin} = \frac{NR}{R + L}$$

A more detailed stochastic model can be developed to capture the transition region between these extremes (cf [6]); the general picture is given by Figure 1. The solid indicates the fraction of time that the processor is busy doing useful work, but does not reflect the increase in cost due to addition of hardware to support more threads. The rate of the cost increase depends on the fraction of the processor that is shared between threads. Since misses occur very late in the pipeline, the only way to achieve a near-zero cycle switch delay is to replicate the entire datapath. A delay of five to ten cycles can be achieved by duplicating only the registers and flushing the pipeline behind the miss.

Factoring in the incremental hardware cost per thread, the gain due to multithreading falls off with increased number of threads. Adding threads has the most impact when the number of threads is small. In this case, the switch delay has little impact, since there are idle cycles in any case, and the chip area devoted to the added threads is small.

In the organization that we are considering, a thread switch is caused by a miss. One expects that increasing the number of threads will increase the miss rate, since the same amount of cache storage is shared by more processes. Increasing the number of threads will decrease the run length and decrease the ratio of R to C. Therefore, with cache effects taken into account the utilization is expected to peak before reaching saturation and drop with increased number of threads, as indicated by lower curve in Figure 1.

Finally, observe that in the linear region the number of outstanding requests being serviced by the memory system at any point is roughly $N - 1$. Thus, to increase the processor utilization the memory system must be enhanced as well.

3 Method of Analysis

To measure the effects of multithreading and timesharing we use a variation on trace driven simulation. The traces are from eight of the SPEC benchmarks: doduc, eqntott, espresso, matrix300,

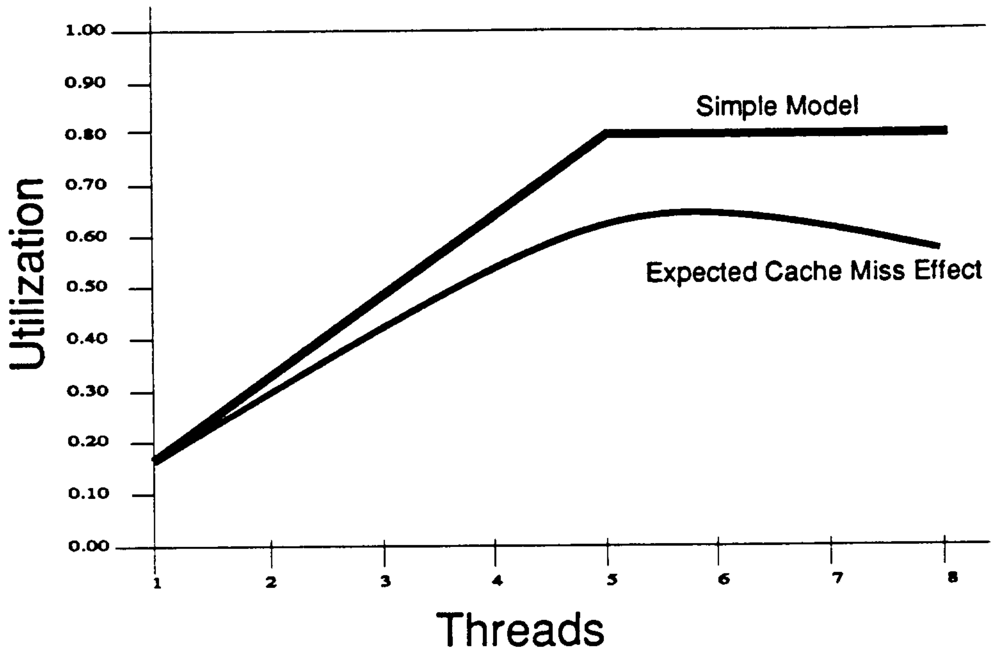


Figure 1: Utilization of a multithreaded processor as a function of the number of threads for latency $L = 100$, switch delay $C = 5$, and average run length $R = 20$.

nasa7, spice2g6, tomcatv, and xlisp. These are the sampled traces of very large reference streams presented in [7]. For each benchmark 50 samples are collected of 200,000 instructions each, giving a total of 10 million instructions per trace. The traces are interleaved to simulate timesharing or multithreading. In the timesharing case the trace is switched on each quantum, i.e., 200,000 instructions are taken from one trace, then 200,000 from the next, and so on in a round robin fashion. For the multithreading case, a trace is followed until a miss occurs, then the next trace is resumed until it misses, and so on.

For the eight-thread studies, all eight programs are used, one for each thread. Two different orderings of the programs were used and the results averaged. For the four-thread studies, four different sets of four programs were used such that each trace is used twice. For the two thread studies, eight different pairs of programs were simulated and averaged. Reference addresses are extended with thread numbers as high-order bits outside the mapped portion of the address.

We assume a lock-up-free on-chip cache able to sustain an arbitrary number of outstanding requests (the number of outstanding requests never exceeds the number of threads). The memory system outside of the cache is modeled as having a latency of a fixed number of cycles (L). Further, we assume that instructions execute in a single cycle. The timeshared system incurs the direct cost of misses, as well as indirect cost of having the cache polluted by other processes on each context switch. For the multithreaded systems, a single cycle is charged for each instruction, plus a constant latency value is charged on each cache miss. The first instruction of the next thread starts C cycles after the miss or when the previous outstanding miss for the that thread completes, whichever is later. We assume that once a value is fetched from main memory the value will be available at least once for the thread which requested it. In effect, the value fetched on a load is provided to the thread that issued the load, allowing the load to complete, as well as being placed in the cache. The load is not reissued, so the thread makes progress even if the fetched line is knocked out of the cache before the thread is resumed. Dirty lines are assumed to be written back to the memory in the background, i.e., perfect write buffering is assumed, so no additional charge is assigned to a miss that replaces a dirty line. For set-associative caches an LRU replacement policy is used.

Our cache simulator is based on Hill's DineroIII, but is almost entirely reworked to support multiple trace streams. At every stage our simulator was verified both by hand and by side-by-side runs against DineroIII. We gather data on cache misses (instruction, data read, and data write), number of context switches, bus traffic (generated from instruction fetches and data reads and writes), number of references to a cache block before it is flushed, and life-time of threads (i.e, how many references occur before a thread is switched). We collected data on caches with different partitioning schemes (N-way partition for N contexts vs. single pool), sizes (1k, 2k, 4k, 8k, 16k, 32k, and 64k bytes), block size of 32 bytes, associativities (1, 2, 4, and 8-way), memory latencies (10, 20, 50, and 100 cycles).

Multiplexing traces raises a question of when to stop the simulation. One approach is to run all traces to completion, representing a fixed work load. The problems with this approach are that differences in trace length essentially operate as weights on the various programs and, more importantly, the number of threads decreases during the last portion of the simulation. The alternative approach is to run for a fixed number of references, repeating traces as necessary so that each program is represented throughout. This represents a fixed segment of a steady-state workload. After examining the data it became clear that the results under the two methods are very different because of the feedback from cache behavior to thread scheduling. Switch-on-miss multithreading introduces unfair scheduling of threads, favoring those with low miss rates. In the following we present results under both approaches. We first adopt the policy that the simulation stops when any trace is exhausted. (Our traces contain the same number of references, to within a few percent.) This reflects the steady-state assumption. We then show how the behavior changes as the rest of the traces complete.

4 Multithreaded Cache Behavior

The most important factor we need to quantify is the performance degradation of the first level cache due to interference among concurrently executing threads. The miss rate determines the thread run length (R), which determines the number of independent threads (i.e., number of concurrent processes in the processor), required to mask a given memory latency and the relative amount of time spent running and switching. The interference among threads is strongly effected by the cache organization. In light of this, we consider a wide range of cache sizes and associativities against various numbers of threads. Note that timesharing causes interference as well, since each quantum steps on the cache footprint established by the previously run processes. Thus, the real question is how the interference under fine-grain switching compares with that of coarser-grain timesharing. Focusing on miss rate at this stage, rather than utilization, has the advantage that memory latency and switch delay can be ignored until the basic cache organization is determined.

Figures 2, 3, and 4 show data miss rates for a variety of cache organizations with two, four, and eight processes on cache ranging from 1KBytes to 64KBytes with associativities of 1, 2, 4, and 8. Miss rates represent the steady-state behavior, *i.e.*, simulation stops when first trace is exhausted. The columns show different degrees of associativity, and the rows show various cache sizes. For each cache size and associativity the data miss rate is shown for timesharing (quantum), multithreading with a single cache pool (pool), and multithreading with the cache partitioned into disjoint equal regions, one per thread. Data for 2-way associative caches of size 32KBytes are shown in bar charts.

There are several interelated factors in the design of a cache to support switch-on-miss multithreading. We proceed by narrowing the space of interesting choices through a series of specific questions.

4.1 Cache size

How large a first level cache is required to support multiple threads? Basically, we need to find the point where the multithreaded miss rate is equal to that under timesharing. For small cache sizes we observe that the multithreading miss rates are higher than those for timesharing. So we should focus our attention on larger cache sizes.

4.2 Cache partitioning

Should the cache be shared among the threads or should threads have separate partitions? Under multithreading, small partitioned caches do slightly better than small pooled caches. For larger caches, a pooled organization is superior. Because the miss rates of a multithreaded C.P.U. using a small cache of either organization is too high to be attractive, we should focus our attention on pooled caches.

4.3 Associativity

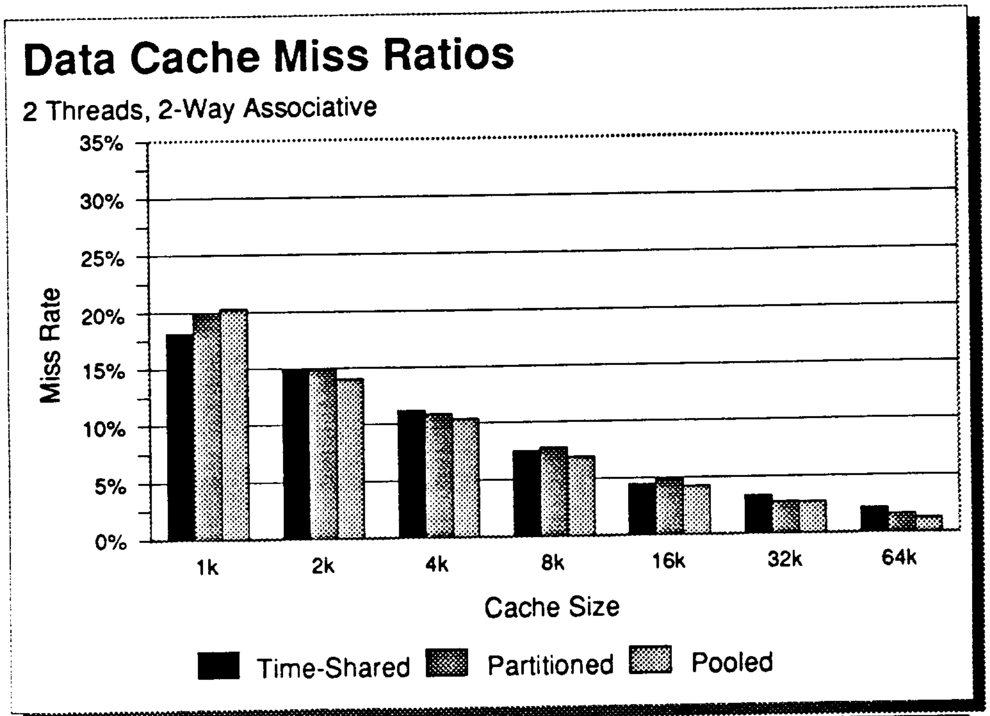
Does multithreading require greater associativity than time sharing? For large pooled caches we see improvements of 30% to 80% from direct-mapped to 2-way associative caches under multithreading. Under timesharing the improvement is substantially less.

Interestingly, large, direct-mapped, partitioned caches outperform large, direct-mapped, pooled caches. Some insight into these effects can be gleaned from Figure 5 which shows the cache footprints of eight processes over time under direct-mapped and 2-way associative caches. With a direct-mapped cache, programs with a broad access pattern, e.g., PID3 matrix mult, are able to hog the cache, whereas with some associativity the amount of storage occupied by the programs is more uniform. Partitioning enforces this uniformity in the direct mapped case. There remains an advantage to pooling, since programs with a very small footprint, e.g., xlisp, are able to make room for the rest.

For large associative caches, the miss rate for the pooled multithreaded cache is very close to that for timesharing; in many cases it is better. This is clearly the design regime to investigate further. Observe that if multithreading does not increase the miss rate, then the increase in memory traffic under multithreading is proportional to the increase in performance. Furthermore, with miss rates in the neighborhood of 2% the thread run length is comparable to the memory access time, so a couple of threads saturate the processor/memory systems. This implies that the number of concurrent miss requests that the memory needs to process is small and also that a very fast switch is unnecessary.

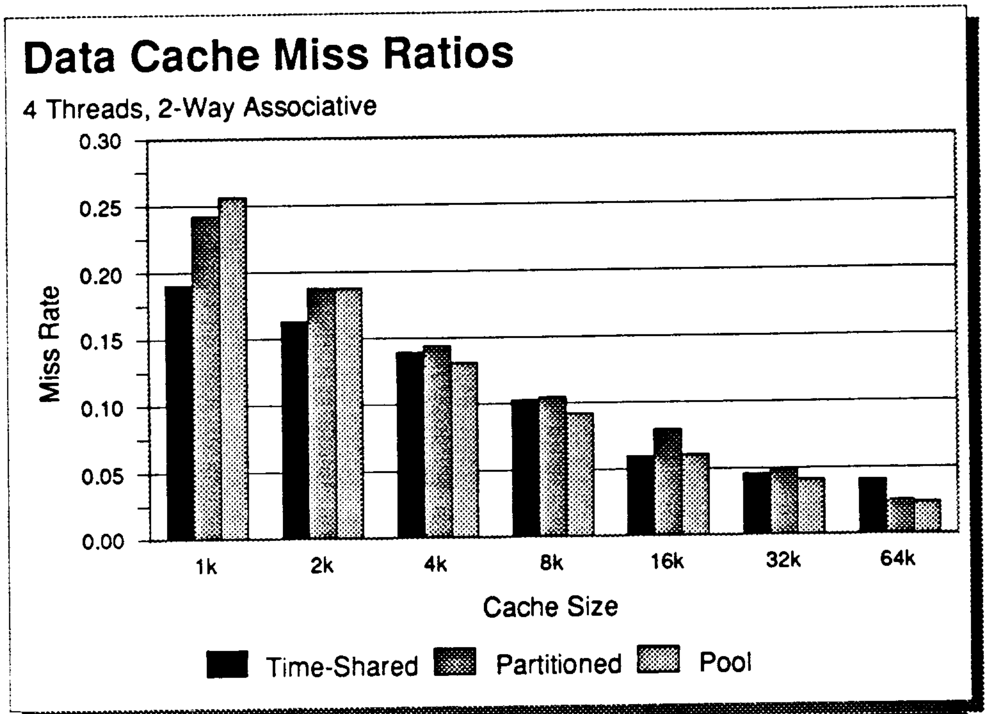
5 Unfairness of Switch-on-miss Multithreading

It is counter-intuitive that multithreading should ever achieve better miss rates than timesharing. Studies to date suggest the opposite[6, 9]. However, the reason for this is quite simple: switch-on-miss multithreading allows threads for processes with low miss rate to run longer. This shows up



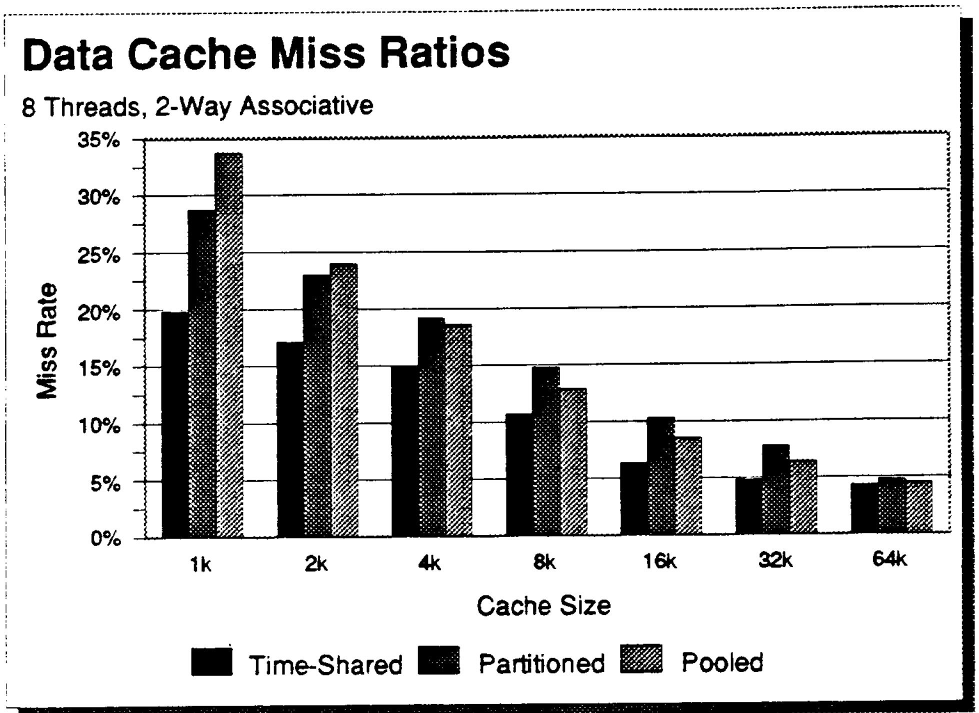
		Assoc1	Assoc2	Assoc4	Assoc8
1k	Quantum	22.30%	18.15%	16.53%	14.10%
	Partitioned	25.44%	19.92%	17.21%	15.27%
	Pooled	25.71%	20.27%	17.56%	15.97%
2k	Quantum	18.37%	14.91%	12.94%	10.60%
	Partitioned	20.02%	14.95%	12.21%	10.07%
	Pooled	19.76%	14.05%	12.02%	9.81%
4k	Quantum	14.15%	11.20%	9.63%	7.85%
	Partitioned	15.73%	10.85%	9.08%	7.55%
	Pooled	14.29%	10.36%	9.11%	7.56%
8k	Quantum	9.45%	7.46%	6.97%	5.86%
	Partitioned	11.03%	7.78%	6.69%	5.64%
	Pooled	10.00%	6.92%	6.01%	5.44%
16k	Quantum	6.82%	4.45%	4.14%	4.24%
	Partitioned	7.56%	4.87%	4.34%	3.83%
	Pooled	7.33%	4.25%	3.43%	3.32%
32k	Quantum	4.87%	3.31%	3.02%	3.01%
	Partitioned	5.18%	2.75%	2.16%	2.09%
	Pooled	4.66%	2.70%	1.92%	1.71%
64k	Quantum	3.29%	2.16%	1.95%	1.54%
	Partitioned	2.96%	1.61%	1.18%	1.06%
	Pooled	2.73%	1.29%	0.59%	0.47%

Figure 2: Data cache miss rates under timesharing, multithreading with a single cache pool, and multithreading with separate partitions for two threads.



		Assoc1	Assoc2	Assoc4	Assoc8
1k	Quantum	23.31%	19.04%	17.37%	15.11%
	Partitioned	30.49%	24.20%	22.79%	21.55%
	Pooled	30.86%	25.65%	23.09%	20.88%
2k	Quantum	19.79%	16.31%	14.39%	12.40%
	Partitioned	24.52%	18.77%	16.09%	14.02%
	Pooled	23.41%	18.76%	16.66%	13.68%
4k	Quantum	16.35%	13.96%	12.41%	10.54%
	Partitioned	19.56%	14.40%	11.57%	9.34%
	Pooled	17.61%	13.15%	11.45%	9.05%
8k	Quantum	11.40%	10.28%	10.67%	8.94%
	Partitioned	15.08%	10.47%	8.88%	7.20%
	Pooled	13.06%	9.22%	8.50%	7.00%
16k	Quantum	8.30%	5.96%	5.76%	6.07%
	Partitioned	11.03%	7.98%	6.70%	5.49%
	Pooled	10.02%	6.07%	5.10%	4.82%
32k	Quantum	6.51%	4.56%	4.28%	4.25%
	Partitioned	7.06%	4.98%	4.32%	3.74%
	Pooled	7.59%	4.16%	3.12%	2.95%
64k	Quantum	5.49%	4.08%	3.95%	3.95%
	Partitioned	4.90%	2.58%	2.07%	2.00%
	Pooled	5.55%	2.43%	1.92%	1.63%

Figure 3: Data cache miss rates under timesharing, multithreading with a single cache pool, and multithreading with separate partitions for four threads.



		Assoc1	Assoc2	Assoc4	Assoc8
1k	Quantum	24.53%	19.80%	17.91%	15.45%
	Partitioned	36.98%	28.68%	26.67%	n/a
	Pooled	39.07%	33.71%	29.46%	27.06%
2k	Quantum	21.08%	17.11%	14.79%	12.66%
	Partitioned	30.76%	22.97%	21.69%	20.00%
	Pooled	29.45%	23.94%	21.09%	18.59%
4k	Quantum	17.66%	14.91%	12.89%	10.87%
	Partitioned	25.07%	19.16%	16.09%	12.90%
	Pooled	23.18%	18.60%	17.25%	11.60%
8k	Quantum	11.82%	10.65%	11.08%	9.23%
	Partitioned	19.71%	14.72%	11.76%	9.17%
	Pooled	17.30%	12.86%	11.41%	8.96%
16k	Quantum	8.60%	6.28%	6.01%	6.24%
	Partitioned	15.32%	10.26%	8.78%	7.07%
	Pooled	13.40%	8.48%	7.22%	6.68%
32k	Quantum	6.84%	4.80%	4.55%	4.51%
	Partitioned	10.45%	7.73%	6.33%	5.36%
	Pooled	10.45%	6.39%	4.97%	4.72%
64k	Quantum	5.84%	4.30%	4.16%	4.15%
	Partitioned	6.66%	4.75%	4.25%	3.69%
	Pooled	8.08%	4.46%	3.14%	3.01%

Figure 4: Data cache miss rates under timesharing, multithreading with a single cache pool, and multithreading with separate partitions for eight threads.

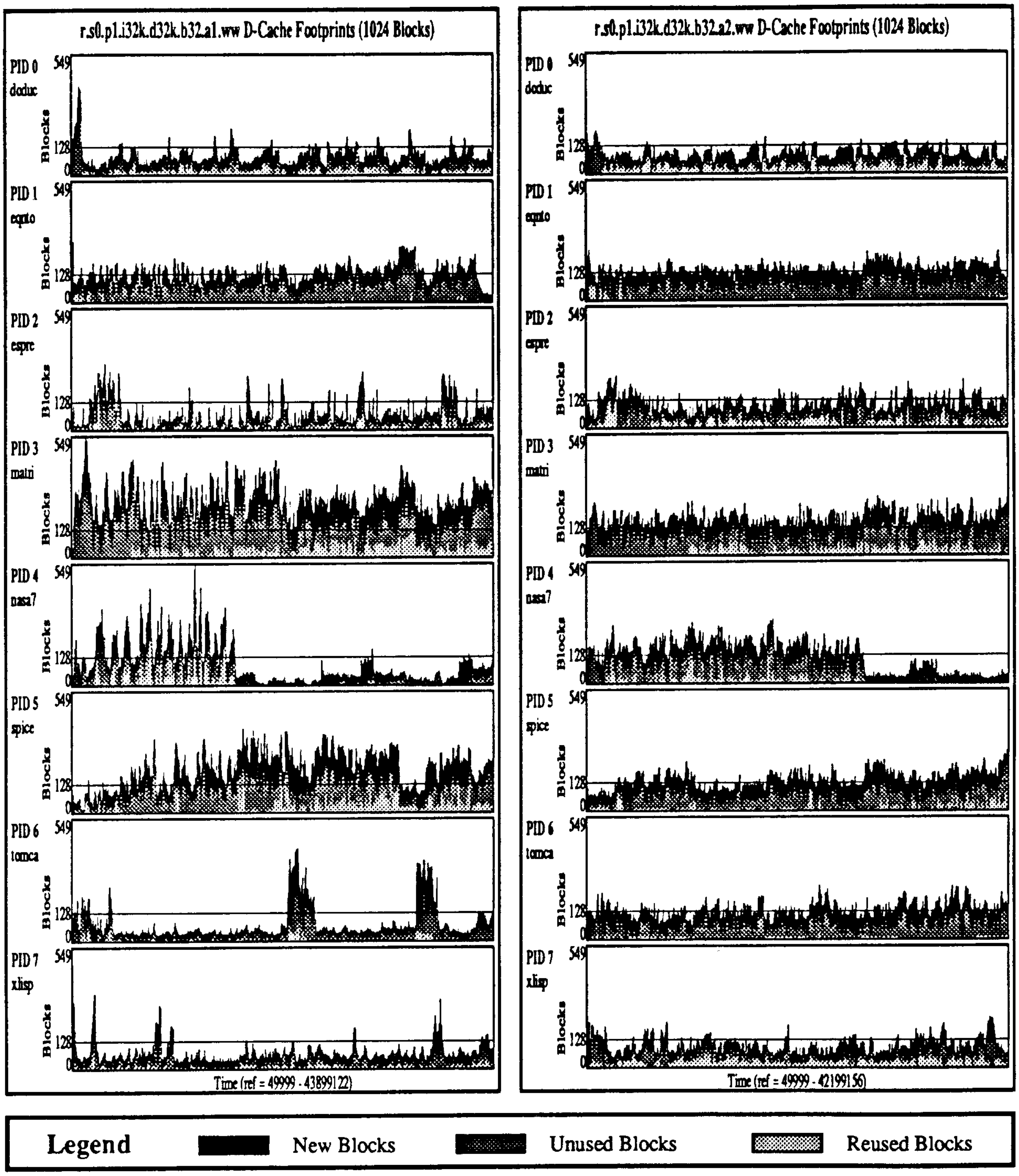


Figure 5: Fraction of the cache owned by each process over time with direct-mapped and 2-way, 32-KByte cache and 8 processes.

quite strongly under our method of analysis. because the simulation stops when the first trace is exhausted. However. the hardware would demonstrate the same unfairness.

To help explain these effects. the upper-left portion of Figure 6 gives data hit ratios for the eight programs in isolation on 4Kyte. 2-way associative cache. The hit ratios are all above 90%. but three of the programs (eqntot. espresso. and xlisp) are in the high 90s. while four programs are in the low 90s. In effect. multithreading and timesharing mix these different reference patterns "on the fly." However. the weights applied to the traces differ. Multithreading gives more processor cycles to those programs with better reference patterns. since missing less means more instructions and more references per turn. Returning to the model of Section 2. the thread length R is related to the miss rate ρ as $R = \frac{1}{\rho}$. In isolation. the miss rates of eqntot (trace2) and spice2g6 (trace6) differ by a factor of 10. The upper-right portion of Figure 6 shows the number of instructions devoted to each process under timesharing and multithreading. Under multithreading. eqntot contributes five times as many instructions as spice2g to the total reference stream. The bottom portion of the figure gives a graphical representation of how hit rate affects the weighting of various traces. Traces with a high hit rate (shown in light grey) make up a larger fraction of the total instruction stream under multithreading and the reverse for traces with a low hit rate.

To understand the miss behavior under a fixed-workload assumption. several configurations were simulated until all the threads completed. Figure 7 show fixed-workload data miss rates for two. four. and eight processes on two-way associative caches ranging from 1KBytes to 64KBytes in size. The threads with low miss rate complete first and make additional room available in the cache for the remaining threads. The footprints and the miss rates change in clearly identifiable segments with each exiting thread. Generally the data miss rate increases as threads exit. because the remaining threads have poorer cache behavior. Figure 8 summarizes this effect for eight threads on a pooled. 32Kbyte. 2-way associative cache. Instruction and data miss rates are given for each individual segment and cumulatively from the beginning of the simulation.

Timesharing based on a quantum of a fixed number of cycles has a similar effect in real machines: a process with fewer misses performs more instructions per second. Thus. in simulating timesharing by multiplexed traces. the traces for the processes with low miss rates get consumed more rapidly that those with high miss rates. In our simulation of timesharing a quantum is 200.000 instructions. regardless of the number of misses. so this effect does not occur. We estimate that it would not be pronounced. as it is under multithreading. because there is no feedback to instruction scheduling.

6 Processor Utilization

To analyze the impact of memory latency and switch delay under multithreading, we fix the cache organization based on results from the previous section. Figure 9 shows the processor utilization under timesharing (quantum) and multithreading for various numbers of processes, with a 32KByte 2-way associative cache. The columns show various memory access latencies, and the rows show various switch delays.

With two threads. 50 cycle memory latency and a switch delay of 5. the cpu utilization under timesharing and multithreading are 62% and 86%. respectively. giving a 35% speedup under multithreading. The speedup increases to 56% with four threads and 52% with eight threads under this latency and delay. When the memory latency increases to 100 cycles, the speedup for two, four. and eight threads is 59%. 112%. and 114%. respectively. Observe, that reducing the switch delay provides little improvement. Increasing the switch delay to 10 has little effect with two threads. because the processor is in the linear regime. However. with four or eight threads the processor is saturated and the drop in utilization with increase in switch cost follows the model presented in

Number of Processor Cycles Devoted to Each Process

Multi-threaded on 32K, 2-way Associative Cache (Pooled)
Hit Ratio of Individual Trace on 4K, 2-way Associative Caches

	Program Name	Hit Rate in Isolation (4k)	Timesharing		Multithreading	
			Number of Instruction	Percent	Number of Instructions	Percent
Trace1	doduc	94.08%	8,067,685	12.18%	2,138,906	6.55%
Trace2	eqntott	99.22%	9,571,743	14.46%	9,999,976	30.60%
Trace3	espresso	98.54%	9,162,432	13.84%	7,103,456	21.74%
Trace4	matrix300	93.42%	7,547,259	11.40%	2,547,673	7.80%
Trace5	nasa7	92.58%	7,543,023	11.39%	2,398,518	7.34%
Trace6	spice2g6	92.01%	8,569,311	12.94%	2,142,514	6.56%
Trace7	tomcatv	92.37%	8,354,713	12.62%	2,776,096	8.50%
Trace8	xlisp	97.66%	7,399,950	11.18%	3,567,492	10.92%

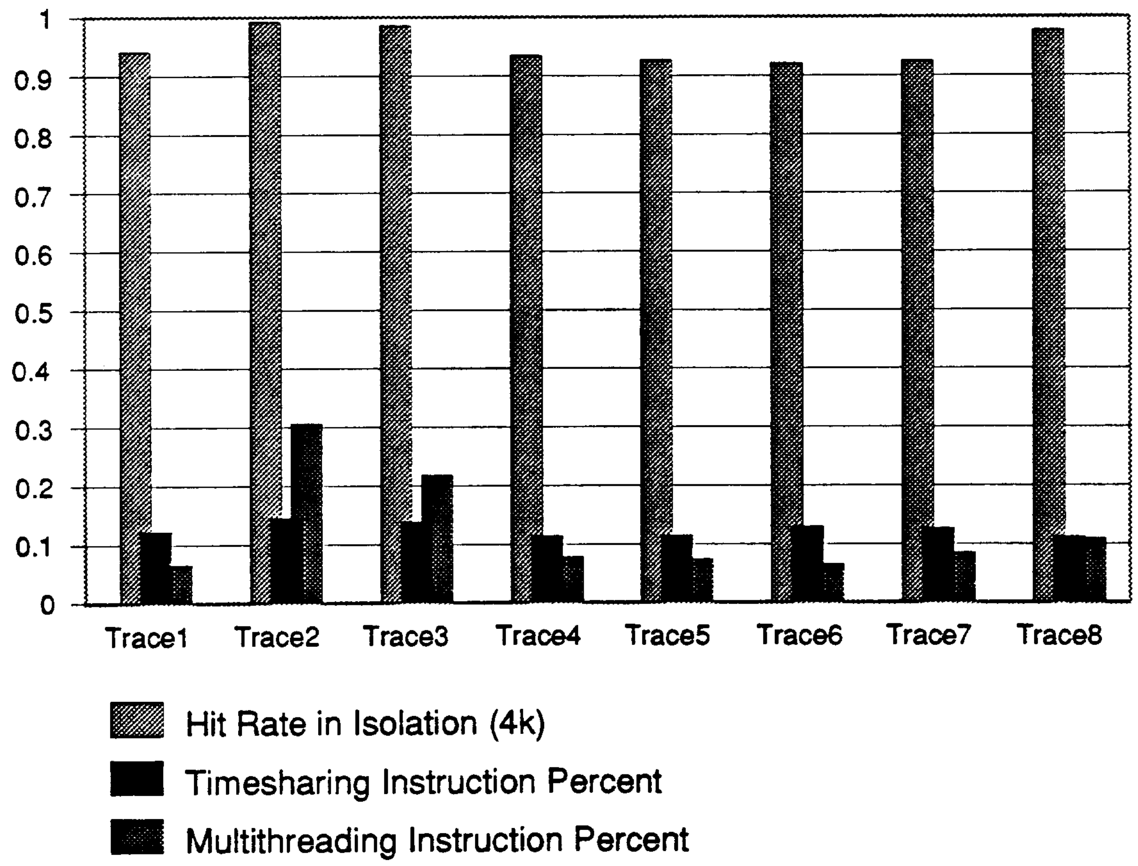
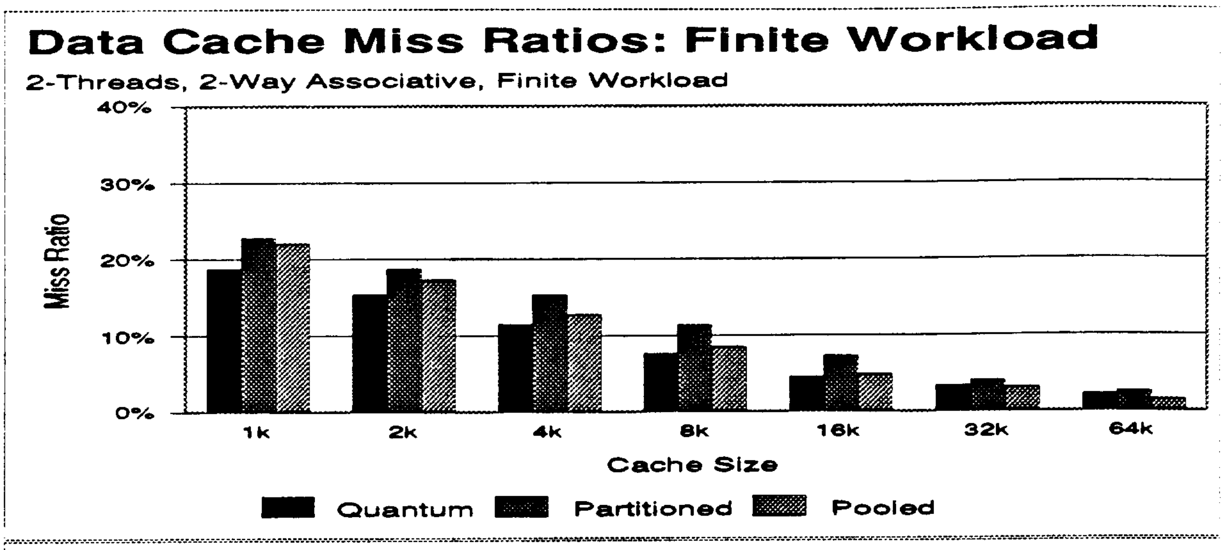
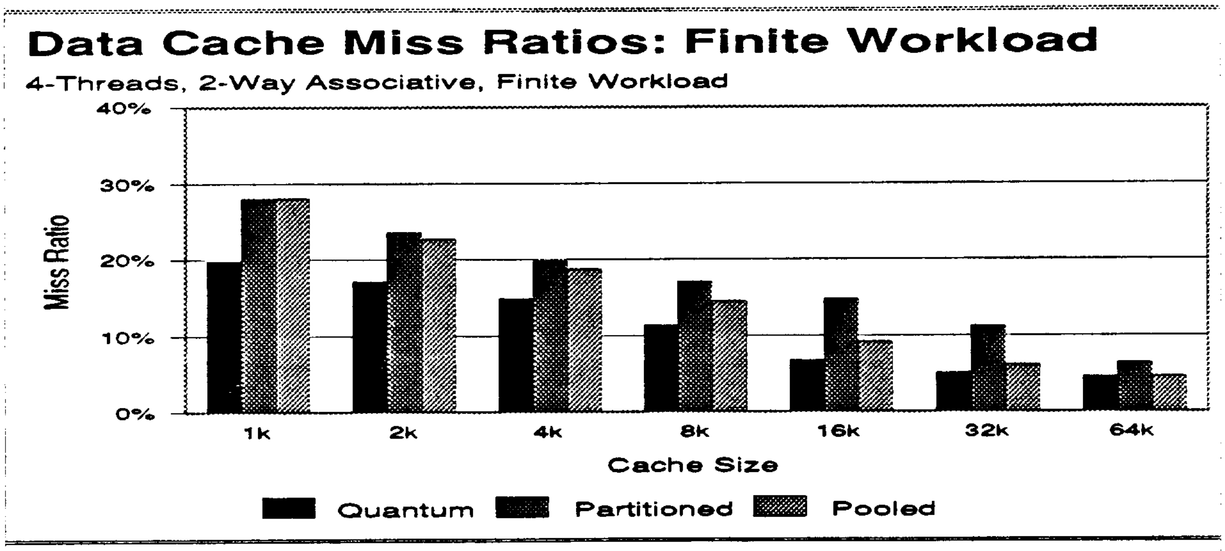


Figure 6: Miss rates in isolation and number of references serviced for each benchmark under timesharing and multithreading





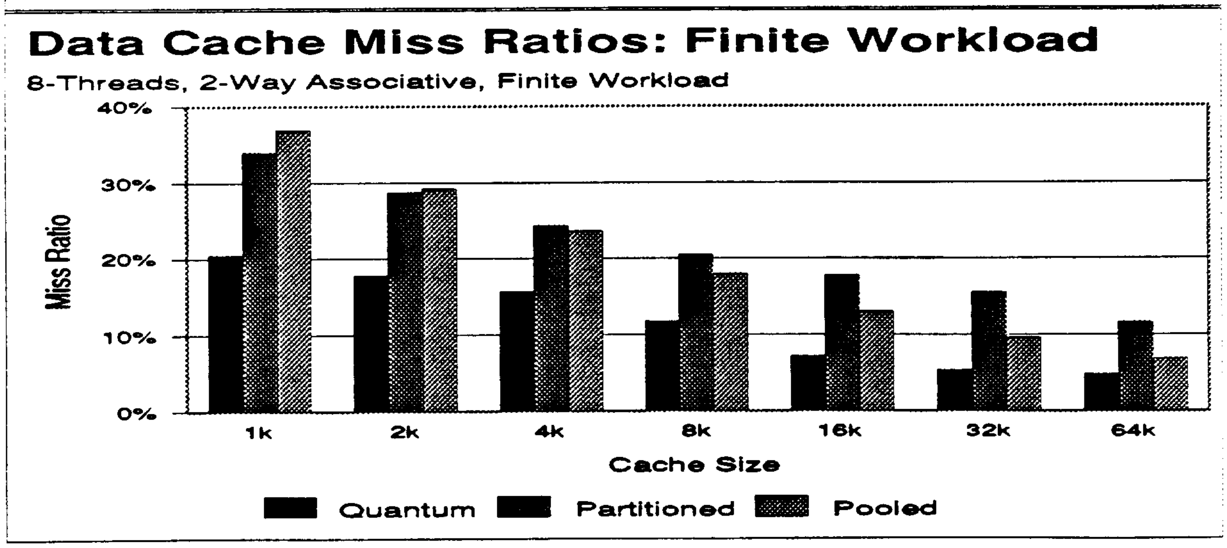


Figure 7: Fixed workoad data cache miss rates under timesharing, multithreading with a single cache pool. and multithreading with separate partitions for two. four. and eight threads.

Segment	Data Miss Rate		Inst. Miss Rate	
	Cumulative %	Segment %	Cumulative %	Segment %
1	6.40	6.40	0.45	0.45
2	6.60	7.20	0.47	0.53
3	6.86	7.49	0.50	0.63
4	6.98	8.20	0.49	0.37
5	7.26	9.97	0.49	0.45
6	9.20	17.05	0.41	0.01
7	9.29	11.50	0.40	0.00
8	9.56	28.98	0.40	0.00

Figure 8: Cumulative and segment miss rates for eight threads on a 32Kbyte, 2-way associative cache as threads exit.

Section 2 very closely. The average run length with four threads is 82 instructions while with eight threads it is 55 instructions.

For a fixed workload with two threads, 50 cycle memory access and a switch delay of 5 the utilization using timesharing and multithreading are 61% and 69%, respectively — a speedup of 13%. The speedup increases to 29% with four threads and 39% with eight threads. When the memory latency increases to 100 cycles, the speedup for two, four, and eight threads is 19%, 44%, and 67%, respectively.

7 Conclusions

As the gap between processor speed and memory cycle time increases, system designers will be faced with a severe challenge in structuring the storage hierarchy. Multithreading uses multiple layers of the hierarchy concurrently, rather than adding more layers. One process uses the processor and cache while another uses the main store. In the presence of large, on-chip caches, the basic instruction processing unit occupies a small fraction of the CPU, so portions of it can be replicated with little marginal cost. The presence of a sizable on-chip cache also makes the investment in a next-level cache less attractive, because it needs to be very large to significantly reduce the average access latency.

In this paper, a argument for limited multithreading was made analytically and validated through trace-driven simulation with trace interleaving. Although there is interference among concurrent threads, we have shown that in a design regime where on-chip caches are large, multithreading does not increase the miss ratio significantly beyond that incurred under timesharing. In fact, multithreading may reduce the observed steady-state miss ratio by favoring processes that operate in cache. The number of threads required is small, since run lengths remain fairly long. Most of the gain is achieved with only two threads, and four appears to be a reasonable upper bound. The demands placed on the cache/memory interface are not great, since it is enough to support a small number of outstanding misses. The increase in memory bandwidth is proportional to the speedup due to multithreading.

One weakness in our method is the use of sampled traces. The sample boundaries are aligned with the timesharing quantum. This favors timesharing, since a process switch would destroy the cache bstate. For multithreading these sample boundaries occur at arbitrary points in the trace,

CPU Utilization	2 Threads	Mem Lat10	Mem Lat20	Mem Lat50	Mem Lat100
SwitchCost 0	Quantum	88.74%	80.00%	62.27%	45.92%
	Pooled	97.85%	95.04%	86.04%	73.38%
SwitchCost 1	Quantum	88.74%	80.00%	62.27%	45.92%
	Pooled	97.28%	94.68%	85.90%	73.35%
SwitchCost 2	Quantum	88.74%	80.00%	62.27%	45.92%
	Pooled	96.61%	94.29%	85.75%	73.32%
SwitchCost 5	Quantum	88.74%	80.00%	62.27%	45.92%
	Pooled	93.84%	92.98%	85.28%	73.21%
SwitchCost 10	Quantum	88.74%	80.00%	62.27%	45.92%
	Pooled	89.51%	89.51%	84.36%	73.02%

CPU Utilization	4 Threads	Mem Lat10	Mem Lat20	Mem Lat50	Mem Lat100
SwitchCost 0	Quantum	88.17%	78.84%	59.84%	42.69%
	Pooled	98.92%	98.88%	97.78%	93.22%
SwitchCost 1	Quantum	88.17%	78.84%	59.84%	42.69%
	Pooled	97.87%	97.86%	96.96%	92.76%
SwitchCost 2	Quantum	88.17%	78.84%	59.84%	42.69%
	Pooled	96.83%	96.83%	96.14%	92.28%
SwitchCost 5	Quantum	88.16%	78.83%	59.84%	42.69%
	Pooled	93.86%	93.86%	93.63%	90.72%
SwitchCost 10	Quantum	88.16%	78.83%	59.84%	42.69%
	Pooled	89.29%	89.29%	89.29%	87.82%

CPU Utilization	8 Threads	Mem Lat10	Mem Lat20	Mem Lat50	Mem Lat100
SwitchCost 0	Quantum	87.90%	78.41%	59.23%	42.08%
	Pooled	98.25%	98.25%	98.23%	97.61%
SwitchCost 1	Quantum	87.90%	78.41%	59.23%	42.08%
	Pooled	96.56%	96.56%	96.55%	96.11%
SwitchCost 2	Quantum	87.90%	78.41%	59.23%	42.08%
	Pooled	94.93%	94.93%	94.92%	94.62%
SwitchCost 5	Quantum	87.90%	78.41%	59.23%	42.08%
	Pooled	90.34%	90.34%	90.34%	90.28%
SwitchCost 10	Quantum	87.90%	78.41%	59.23%	42.08%
	Pooled	83.61%	83.61%	83.61%	83.61%

Figure 9: Processor Utilization under timesharing and multithreading with various number of threads. Average of runs with two, four, and eight contexts.

thereby introducing a flurry of misses. To eliminate sampling for benchmarks of this size. it would be necessary to avoid storing traces altogether.

Given current growth rates. the point where multithreading appears to be superior lies not far into the future. These results need to be validated with studies based directly on interactive workloads with operating system effects and limits on memory system capability. Nonetheless. the results presented here indicate that a detailed engineering analysis of multithreading is warranted. perhaps even in the next generation of microprocessors. Changes in processor design along these lines may yield a better building block for large-scale multiprocessors as well. However. in shared-memory multiprocessors one cannot expect miss rates to drop arbitrarily. since misses reflect the inherent communication between portions of the computation. Multithreading in the uniprocessor case is particularly attractive because thread run lengths become long with caches that can reasonably be expected to reside on the CPU chip.

Acknowledgements

The traces. which were generously provided by Klaus Schauser. were generated on a MIPS R2000 using pixie and pixtrace. They were converted to dinero format by the program pixtodin and from there to our more compact csim format. Computing resources were graciously provided by the Sprite project at UC Berkeley.

This work was supported by National Science Foundation PYI Award (CCR-9058342) with matching funds from Motorola Inc. and the TRW Foundation.

References

[1] A. Agarwal. B. Lim. D. Kranz. and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*. pages 104–114. Seattle. Washington. May 1990.

[2] Arvind and D. E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*. volume 1. pages 225–253. Annual Reviews Inc.. Palo Alto. CA. 1986. Reprinted in Dataflow and Reduction Architectures. S. S. Thakkar. editor. IEEE Computer Society Press. 1987.

[3] R. H. Halstead. Jr. and T. Fujita. MASA: a Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proc. of the 15th Int. Symp. on Comp. Arch.*. pages 443–451. 1988.

[4] H. F. Jordan. Performance Measurement on HEP — A Pipelined MIMD Computer. In *Proc. of the 10th Annual Int. Symp. on Comp. Arch.*, Stockholm, Sweden, June 1983.

[5] D. Lenoski. J. Laudon. K. Gharachorloo. A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, pages 148–159, Sealttle, Washington, May 1990.

[6] R. Saavedra-Barrerra. D. E. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the 2nd Annual Symp. on Par. Algorithms and Arch.*, July 1990.

[7] K. E. Schauser. K. Asanovic. D. A. Patterson. and E. H. Frank. Evaluation of a Stall Cache: An Efficient Restricted On-Chip Instruction Cache. Technical Report UCB/CSD 91/641, Univ. of Calif.. Berkeley. Computer Science Div.. July 1991.

[8] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In *Proc. of Supercomputing '88*, pages 35–41, Orlando, FL, 1988.

[9] W. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proc. of the 16th Int. Symp. on Comp. Arch.*, pages 273–280. Jerusalem, Israel, May 1989.