Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design

Based on papers by:
A.Fedorova, M.Seltzer, C.Small, and D.Nussbaum

Pisa – November 6, 2006

Introduction

Multithreaded Chip Multiprocessors (CMT) are a new generation of processors designed to improve performance of memory–intensive applications.

The goal of CMTs is to improve performance of modern applications such as Web Services, application servers and on–line transaction processing systems.

- multiple threads executing short sequences of integer operations, with frequent dynamic branches
- low cache locality and branch prediction accuracy
- low utilization of the processor pipeline

Introduction

CMT processors combine:

- chip multiprocessing (CMP)
- hardware multithreading (MT)

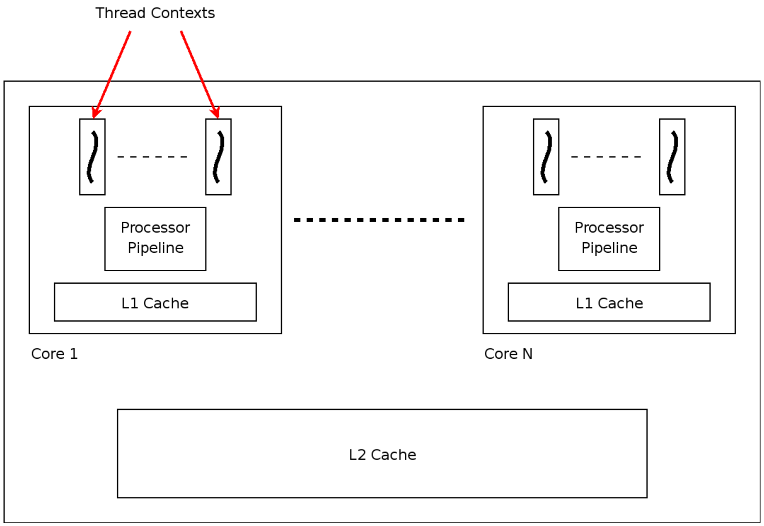
A CMP processor include multiple processor cores on a single chip, which allows more than one thread to be active at a time.

An MT processor interleaves execution of intructions from different threads. As a results, if one thread blocks on a memory access, other threads can make forward progress.

As a consequence, CMTs are equipped with dozens of simulatneously active thread contexts.

- Competition for shared resources is high!

The Structure of a CMT Processor



Examples of CMT Processors

INTEL

- Core Duo: 2 cores, ... x2 L1 Cache, 2MB L2 Cache, 1.66–2.13 Ghz
- Core 2 Duo: 2 cores, ... x2 L1 Cache, 4MB L2 Cache, 1.86–2.93 Ghz
- ...

AMD

- Athlon 64 X2: 2 cores, 128KB x2 L1 Cache, 2MB L2 Cache, 2.00–2.60 Ghz

SUN

- UltraSPARC T1 (Niagara): up to 8 cores, 24KB x8 L1 Cache, 3MB L2 Cache, 1.00–1.20Ghz

Thread Interleaving in Each Processor Core

Two approaches exists to handle thread interleaving in a CPU core

- **Coarse–grained multithreading** switches to a new thread when a thread occupying the processor blocks on a memory request
  - High context switch cost! (the decision to switch depends on determining whether a cache miss occurred, and this is made late in the pipeline)
- **Fine–grained multithreading** switches threads on every cycle
  - The performance of a single thread is extremely poor

CMT processors usually realize fine–grained multithreading.

Plan of the Talk

How contention for

- the processor pipeline
- L1 data chache
- L2 cache

affects system performance? Which of these shared resource may become performance bottlenecks?

A new scheduling algorithm to improve L2 performance.

A CMT System Simulator

To study contention for shared resources, a simulator of a CMT has been developed as a set of exetension to the Simics simulation toolkit.

- simulator based on an UltraSPARC II machine
- the simulated machine can bootstrap the Solaris OS and a standard Unix environment
- the number of CPU cores and the number of thread contexts for each core are configurable
- cache sizes, cache latencies and memory latency are configurable

Pipeline Contention

Threads differ in how they use the processor pipeline.

- **Compute–intensive** threads issue instructions freqently and utilize the pipeline intensively
- **Memory–intensive** threads frequently stall while waiting for a response from the memory hierarchy

The scheduler can balance the demand for pipeline resources across cores by co–scheduling compute–intensive threads with memory–intensive threads on the same processor core.

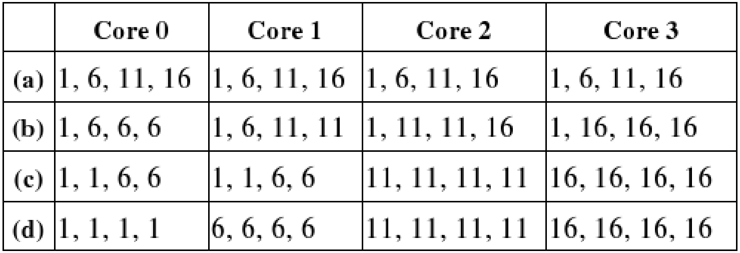
A scheduler can identify compute–intensive and memory–intensive threads by measuring the workload's CPI (cycles per instruction) metric.

Pipeline Contention

Experiment to evaluate the potential performance improvement obtained by using a scheduler that co–schedules compute–intensive threads with memory–intensive threads.

- On a machine with 4 cores and 4 thread contexts for each core
- Tried several ways to schedule 16 threads
  - 4 each with CPIs 1, 6, 11, and 16
- The result is the measure of IPC (Instructions per Cycle) achieved by each schedule

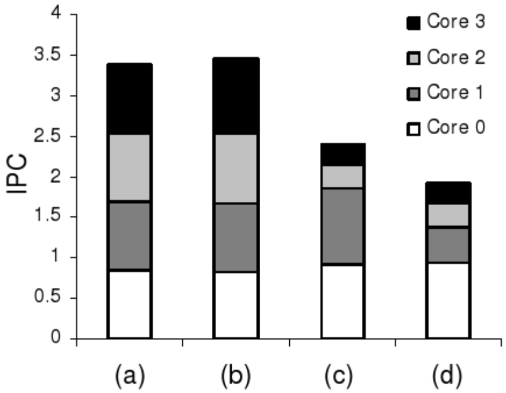
Pipeline Contention



	Core 0	Core 1	Core 2	Core 3
(a)	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16	1, 6, 11, 16
(b)	1, 6, 6, 6	1, 6, 11, 11	1, 11, 11, 16	1, 16, 16, 16
(c)	1, 1, 6, 6	1, 1, 6, 6	11, 11, 11, 11	16, 16, 16, 16
(d)	1, 1, 1, 1	6, 6, 6, 6	11, 11, 11, 11	16, 16, 16, 16

Assignment of threads to cores. Schedules (a) and (b) match compute–intensive threads with memory–intensive threads. Schedules (c) and (d) place compute–intensive threads on the same core.

Pipeline Contention



IPC achieved by each schedule.

Pipeline Contention

While the potential performance improvement from this scheduling technique is dramatic, to achieve it, it was necessary to have threads with a wide range of CPIs.

What happens with real workloads?

- Average CPIs for a number of standard integer benchmarks have been measured.
    - SPEC CPU, SPEC JVM, SPEC JBB, and SPEC Web
- For most benchmarks, the average CPI is around 4
    - All between 2.37 and 5.11

Other experiments performed with SPEC benchmarks confirmed that performance gains from CPI-based scheduling are modest (about 5%) for such workloads.

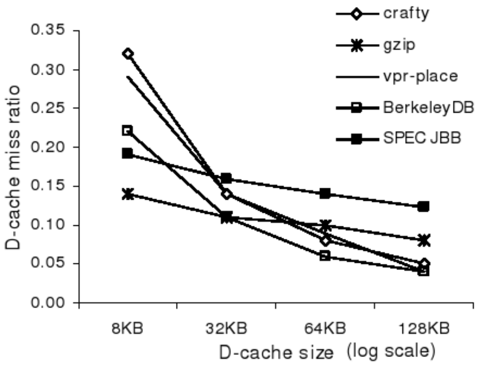
The L1 Data Cache

Multithreaded processors are typically configured with small L1 data caches. This could result in high latencies associated with handling cache faults.

Experiment to evaluate the potential loss of performance due to high L1 data miss rates:

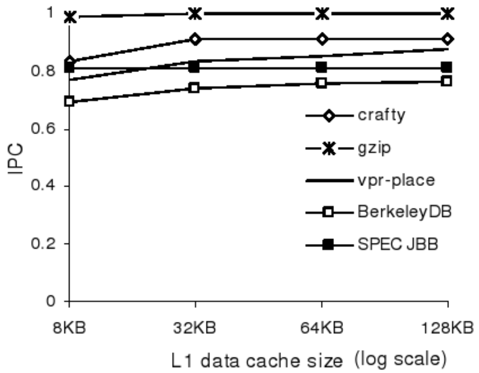
- On a machine with a single core and 4 thread contexts
- Executed four copies of some benchmarks by varying the cache size
- The result is the measure of the cache miss ratios and IPCs achieved by each group of benchmarks, for each cache size

The L1 Data Cache



L1 data cache miss ratios as the cache size changes.

The L1 Data Cache



Pipeline utilization as the cache size changes.

The L2 Cache

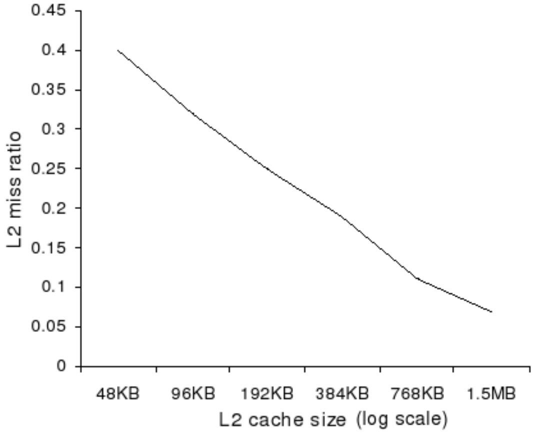
The L2 cache has a greater potential for becoming a performance bottleneck because the latency between the L2 and main memory is very high.

- In a Intel P4-HT, L1 $\longrightarrow$ L2 costs 18 cycles, and L2 $\longrightarrow$ Mem costs 360 cycles

Experiment to evaluate the effect of L2 performance.

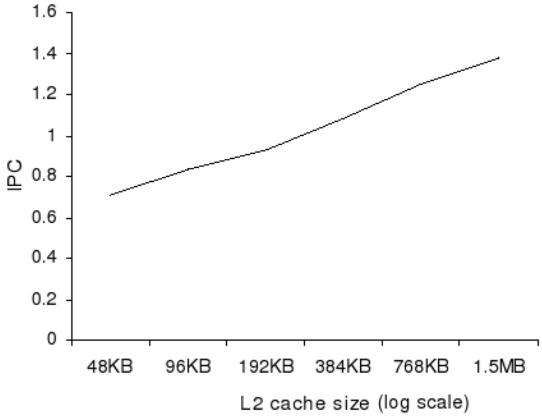
- On a 2 core CPU with 4 thread contexts and 8Kb of L1 cache
- Chosen 9 benchmarks with good and poor cache locality
- Executed two copies of each benchmark (18 threads) by varying the cache size
- Thread scheduled by the standard Solaris scheduler!
- The result is the measure of the cache miss ratios and IPCs achieved for each cache size

The L2 Cache



L2 miss ratios for the 18–thread SPEC workload.

The L2 Cache



IPC for the 18–thread SPEC workload.
Processor IPC is sensitive tho the L2 miss ratio.

The L2 Cache

Performance degradation is evident as the L2 cache becomes smaller

$+$

Modern applications exhibit a dangerous trend of becoming progressively more data–intensive

$+$

Changing software is easier than changing hardware

$=$

It is wise to equip the OS with the ability to handle L2 cache shortage

Balance–Set Scheduling

Originally proposed by Denning as a way of improving the performance of virtual memory.

It is based on the notion of *Working Set*: the data that must be present in main memory (or in cache) to assure the efficient execution of a program (a thread).

The idea is to

1. separate all runnable threads in groups such that the combined working set of each group fits in the cache
2. schedule a group at a time for the duration of a time slice

Balance–Set Scheduling

Problem: experiments show that working set size is not a good indicator of the workload's cache behavior.

- some benchmarks with large working sets produced lower miss ratios than others with a small working set

Working set size is a good indicator only if the program access its working set uniformly!

A better way to asses a workload's cache behavior is necessary.

The Reuse–Distance Model

Proposed by Berg and Hagersten.

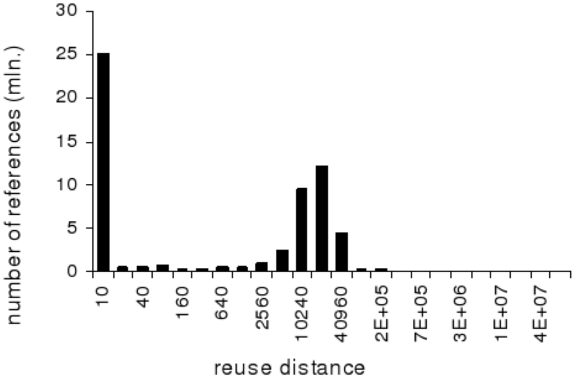
It is based on the notion of *Reuse Distance* of a memory location: the amount of time (num. of memory references) that passes between successive references to the same memory location.

- The smallest is the reuse distance of a location, the greater is the probability that a reference will result in a hit.

The input necessary for this model is the *reuse–distance histogram*: it counts the total number of re–references that fell within each distance.

- Can be built at runtime by using the standard hardware watchpoint mechanism.

The Reuse–Distance Model



An example of reuse–distance histogram (188.ammp).

The Reuse–Distance Model

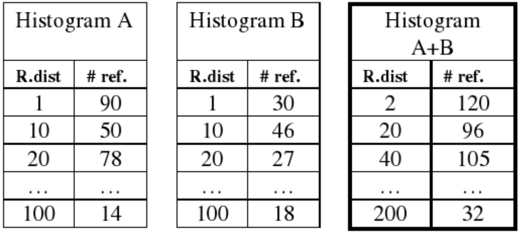
Given a reuse–distance histogram for a single thread, the reuse–distance model estimates a cache miss ratio.

For balance–set scheduling, we need to estimate cache miss ratios for groups of threads. Two methods:

- COMB
- AVG

The Reuse–Distance Model

- **COMB**: (i) sum the number of references for each reuse distance in each histogram, (ii) and multiply each reuse distance by the number of threads in the group, (iii) apply the reuse–distance estimation on the resulting histogram.



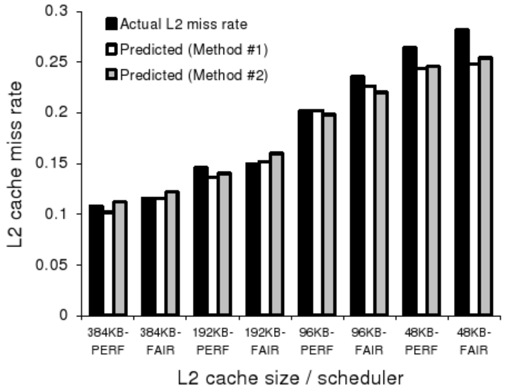
Histogram A	
R.dist	**# ref.**
1	90
10	50
20	78
...	...
100	14

Histogram B	
R.dist	**# ref.**
1	30
10	46
20	27
...	...
100	18

Histogram A+B	
R.dist	**# ref.**
2	120
20	96
40	105
...	...
200	32

- **AVG**: (i) assume that each thread runs with its own dedicated partition of a cache, (ii) estimate ratios for individual threads, (iii) compute the average.

The Reuse–Distance Model



Actual vs predicted miss rates (#1=COMB, #2=AVG)

The Reuse–Distance Model

Estimed miss ratios are, on average, within 17% of the actual ones

- However, both COMB and AVG are "accurate enough" to distinguish between thread groups that produce high miss ratios and those that produce low miss ratios

The drawback of COMB is that it is computationally too expensive to implement on a real system.

- In a machine with 32 thread contexts and 100 threads the scheduler has to combine $\binom{100}{32}$ histograms!

AVG wins!

The Scheduling Algorithm

Step 1 Computing miss rate estimations (Periodically)

- Assume N runnable threads and M hardware thread contexts
- Compute all the miss rate esimations of the $\binom{N}{M}$ groups of M threads by using the reuse–distance model and AVG

The Scheduling Algorithm

Step 2 Choosing the L2 miss ratio threshold (Periodically)

- The scheduler will keep the L2 miss ratio below this threshold
- The goal is to set this threshold to be as low as possible
- Analysis of the individual reuse–distance histograms allows to identify the most cache–greedy thread
- Once the scheduler has computed the estimated miss ratios for all possible groups of threads, it picks the smallest miss ratio among the groups including the greediest thread
- The picked ratio is the threshold

The Scheduling Algorithm

Step 3 Identify the groups that will produce low cache miss ratios (Periodically)

- Simply discard the groups of threads whose estimated miss ratio is above the threshold
- The remaining groups are *candidate groups*
- Every runnable thread is at least in one candidate group!

The Scheduling Algorithm

Step 4 Scheduling decision (Every time a time slice expires)

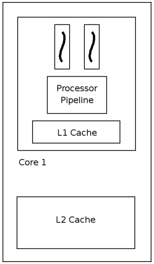
- Choose a group from the set of candidate groups
- Schedule the threads in the group to run during the current time slice
- Keep track of how much processor time each thread has received

The Scheduling Algorithm

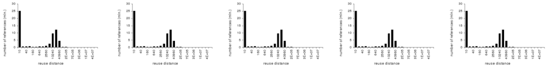
To choose thread groups there can be two policies: performance–oriented (PERF) and fairness–oriented (FAIR)

- With PERF, we select the group with the lowest miss ratio and containing threads that have not yet been selected, until each thread is represented in the schedule
- With FAIR, we select the group with the greatest number of the least frequently selected threads

The Scheduling Algorithm: An Example



Single core CPU
2 thread contexts

5 running threads	T1	T2	T3	T4	T5
reuse-distance histograms					
estimated cache miss rate with 1/2 L2 cache	0.1	0.2	0.1	0.15	0.05

most cache-greedy thread

$\binom{5}{2}$ groups of 2 threads

[T2,T4]	=	0.175
[T1,T2]	=	0.15
[T2,T3]	=	0.15
[T2,T5]	=	0.125
[T1,T4]	=	0.125
[T3,T4]	=	0.125
[T4,T5]	=	0.1
[T1,T3]	=	0.1
[T1,T5]	=	0.075
[T3,T5]	=	0.075

threshold

candidate groups

PERF : [T3,T5] , [T1,T5] , [T4,T5] , [T2,T5] , [T3,T5] , [T1,T5] , ...

FAIR : [T3,T5] , [T1,T4] , [T2,T5] , [T3,T5] , [T1,T4] , [T2,T5] , ...

Emulating the scheduler

Experiment performed by using the 18–thread SPEC workload described before on a dual core CPU with 4 thread contexts on each core.

Reuse–distance histograms computed by the simulator for all threads off–line

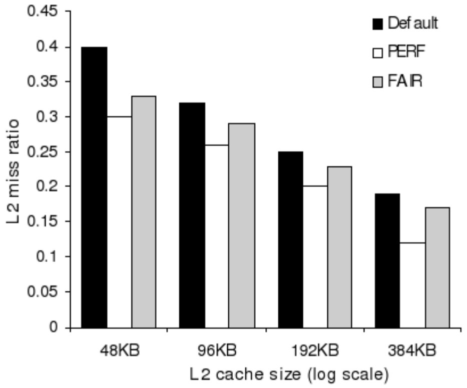
- no information about the overhead!

Examined all the $\binom{18}{8}$ combinations of threads and computed the candidate set.

From the candidate set picked several groups using either PERF or FAIR policy to obtain the final *schedule*.

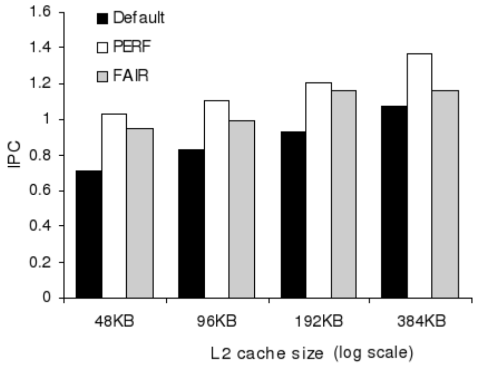
Executed the schedule on the simulator.

Emulating the Scheduler



L2 miss ratios achieved with the default (Solaris) scheduler, and the balance set scheduler.

Emulating the Scheduler



IPC achieved with the default (Solaris) scheduler, and the balance set scheduler.

Conclusions

The authors determined that contention for L2 cache has the greatest effect on system performance.

With balance–set scheduling, the L2 miss rates has been reduced by 19–37% when using the PERF policy and 9–18% when using the FAIR policy

The same performance improvements achived with PERF can be achieved by using the defauld scheduler by doubling the L2 cache size.