

# Understanding Cache Interference

by

**M.W.A. Settle**

B.A., University of Colorado , 1996

M.S., University of Colorado, 2001

A thesis submitted to the  
Faculty of the Graduate School of the  
University of Colorado in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
Department of Electrical and Computer Engineering  
2006

This thesis entitled:  
Understanding Cache Interference  
written by M.W.A. Settle  
has been approved for the Department of Electrical and Computer Engineering

---

Daniel A. Connors

---

Prof. Andrew Plesczkun

---

Prof. Dirk Grunwald

Date \_\_\_\_\_

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Settle, M.W.A. (Ph.D., Electrical Engineering)

Understanding Cache Interference

Thesis directed by Prof. Daniel A. Connors

## Abstract

System performance is increasingly coupled to cache hierarchy design as chip-multiprocessors (CMPs) increase in core count across generations. Higher core counts require large last level cache (LLC) capacities to avoid costly off-chip memory bandwidth and the inherent bottleneck of memory requests from multiple active cores. Currently there are two divisions of thought in CMP cache design- shared versus private last level cache. At center of the issue is that CMP systems can improve different workloads: the throughput of multiple independent single-threaded applications and the high-performance demands of parallel multi-threaded applications. Consequently, maximizing the improvement of CMP performance in each of these domains requires opposing design concepts. As a result, it is necessary to investigate the behaviors of both shared and private LLC design models, as well as investigate an adaptive LLC approach that works for multiple workloads.

This paper proposes a scalable CMP cache hierarchy design that shields applications from inter-process interference while offering a generous per core last level cache capacity and low round trip memory latencies. A combination of parallel and serial data mining applications from the Nu-Mine Bench suite along with scientific workloads from the SpecOMP suite are used to evaluate the cache hierarchy performance. The results show that the scalable CMP cache hierarchy decreases the average memory latency of the parallel workloads by 45% against the private cache configuration and an average of 15% against the shared cache. In addition, the memory bandwidth is 25% lower than the private cache bandwidth for parallel applications and 30% lower for the serial workloads.

## Dedication

To all of the fluffy kitties.

## Acknowledgements

Here's where you acknowledge folks who helped.

## Contents

### Chapter

<b>1</b>	Introduction	1
1.1	Current CMP Cache Architectures . . . . .	2
1.2	Industry Trends in CMP Cache Architectures . . . . .	4
1.3	Cache Hierarchy Design Considerations . . . . .	6
1.3.1	Latency . . . . .	6
1.3.2	Capacity . . . . .	6
1.3.3	Interference . . . . .	7
1.4	Workload Dependence . . . . .	8
1.4.1	Dynamic Cache Organization . . . . .	10
1.5	Thesis Statement Contributions and Outline . . . . .	10
<b>2</b>	Measuring Shared Cache Interference	12
2.1	Capacity vs. Conflict Misses . . . . .	14
2.1.1	Replacement Algorithms . . . . .	16
2.1.2	LRU Stack Hit Order . . . . .	19
2.2	Performance Monitoring Units (PMUs) . . . . .	22
2.2.1	Fine Grain Access Counters . . . . .	23
2.2.2	Activity Vectors . . . . .	24
2.3	Shared Cache Performance Metrics . . . . .	26
2.3.1	Inter-thread Eviction Counters . . . . .	28
2.3.2	High Level Counters . . . . .	29
2.4	Interference Prediction . . . . .	30
2.4.1	Relative Access Rate . . . . .	31
2.4.2	Local Miss Rate . . . . .	33
2.4.3	LRU Stack Rank . . . . .	34

2.4.4	Linear Regression Correlation . . . . .	34
<b>3</b>	<b>Dynamic Cache Management System</b>	<b>36</b>
3.1	Cache Partitioning Description . . . . .	37
3.2	Physical Partitions . . . . .	38
3.3	Virtual Partitions . . . . .	39
3.4	Partition Management . . . . .	39
3.4.1	Precision Model . . . . .	40
3.4.2	Shadow Cache . . . . .	41
3.4.3	Decreasing ways . . . . .	43
3.5	Hardware Cache Partitioning Algorithms . . . . .	43
3.6	Fairness . . . . .	46
3.7	Algorithms . . . . .	47
3.7.1	Heuristic Based HW Cache Partitioning Algorithm . . . . .	47
3.8	Hardware Only Performance results . . . . .	49
3.8.1	Reuse . . . . .	53
3.8.2	Cache Partitioning . . . . .	54
3.9	Grand Unified Theory Algorithm . . . . .	54
3.10	Summary . . . . .	55
<b>4</b>	<b>A Scalable CMP Cache Hierarchy</b>	<b>56</b>
4.1	Introduction . . . . .	56
4.2	Workloads . . . . .	58
4.2.1	Parallel Workloads . . . . .	58
4.3	Scalable LLC . . . . .	60
4.4	Parallel Application Memory Latency and Bandwidth . . . . .	61
4.5	Serial Application Latency and Bandwidth . . . . .	64
4.6	Workload Analysis . . . . .	66
4.7	Parallel Workloads . . . . .	66
4.7.1	Serial Workloads . . . . .	67
4.7.2	Fairness . . . . .	68
4.8	Summary . . . . .	69

<b>5</b>	<b>Experimental Method</b>	<b>70</b>
5.1	Simulation Infrastructure . . . . .	70
5.2	Workloads Studied . . . . .	71
5.3	Real Hardware Measurements . . . . .	72
5.4	Performance Monitors . . . . .	72
5.5	OS Scheduler Modifications . . . . .	73
5.6	Linux scheduler implementation . . . . .	74
<b>6</b>	<b>Related Work</b>	<b>78</b>
6.1	CMP Cache Hierarchies . . . . .	78
6.2	Innovative Uniprocessor Cache Organizations . . . . .	80
6.3	Cache Sharing . . . . .	80
6.4	OS Scheduling For MT Performance . . . . .	82
6.5	Transition to Multithreaded Processors . . . . .	82
6.5.1	Course Grain Multithreading . . . . .	82
6.5.2	Fine Grain Multithreading . . . . .	83
6.5.3	SMT Simultaneous Multithreading . . . . .	84
6.5.4	Performance Issues . . . . .	84
6.5.5	Compilers . . . . .	84
	<b>Bibliography</b>	<b>86</b>

## Tables

### Table

2.1	Working Set Sizes . . . . .	20
2.2	Important Interference Variables . . . . .	32
4.1	OpenMp Application Working Set Size . . . . .	59
4.2	Unloaded Cache Access Latencies . . . . .	61
4.3	Serial Workload Name Key . . . . .	64
5.1	Memory Simulator Configuration . . . . .	70
5.2	Working Set Sizes . . . . .	72

## Figures

### Figure

1.1	Shared Cache Architectures . . . . .	2
1.2	Cache Latency vs. Capacity . . . . .	3
2.1	Miss Rate Dependency . . . . .	13
2.2	Capacity vs. Conflict Hit Rate . . . . .	15
2.3	LRU Stack Abstraction . . . . .	16
2.4	Cache Address Fields . . . . .	17
2.5	Line Address Reuse . . . . .	18
2.6	Early Eviction Optimization . . . . .	18
2.7	LRU Stack for All Bmarks . . . . .	21
2.8	GCC Temporal LRU Stack . . . . .	21
2.9	Twolf Hit Distribution Across Sets . . . . .	24
2.10	GCC Twolf Per Set Miss Counts . . . . .	25
2.11	GCC Per Set Miss Counts . . . . .	25
2.12	Illustrating the basic construction of activity vectors. . . . .	27
2.13	Inter-thread Interference Graph . . . . .	29
2.14	Eclat Access and LRU . . . . .	31
2.15	Interference Counter Correlation . . . . .	35
3.1	Partitioned Cache . . . . .	38
3.2	Model Checker . . . . .	40
3.3	Shadow cache . . . . .	41
3.4	Shadow Cache Performance . . . . .	43
3.5	gzip_mcf Optimized Cache Line Allocation . . . . .	44
3.6	Cache Sharing Performance . . . . .	45

3.7	Cache Sharing Fairness . . . . .	47
3.8	L2 Cache Allocation Model Speedup . . . . .	50
3.9	L2 Cache Fairness . . . . .	51
3.10	L2 Cache Hit Count . . . . .	51
3.11	2MB L2 Cache Speedup . . . . .	52
3.12	CMP L2 Cache Hit Count . . . . .	52
3.13	Partition Algorithm State Transitions . . . . .	54
4.1	Scalable CMP Cache Hierarchy . . . . .	61
4.2	Average Memory Latency Across LLC Organizations . . . . .	62
4.3	Memory Bandwidth Parallel Applications Across LLC Organizations . . . . .	63
4.4	Memory Latency of Serial Applications Across LLC Organizations . . . . .	64
4.5	Memory Bandwidth of Serial Applications Across LLC Organizations . . . . .	66
4.6	L2 Access Components . . . . .	66
4.7	4M Shared LLC Cache Fairness . . . . .	68
4.8	4M Scalable LLC Cache Fairness . . . . .	69
5.1	Default linux activity aware scheduling opportunity. . . . .	77

## Chapter 4

### A Scalable CMP Cache Hierarchy

#### 4.1 Introduction

Modern CMP processors are effectively multiprocessor systems on a single chip. As a result they are expected to achieve comparable performance to traditional MP systems composed of arrays of uniprocessor chips. The fundamental difference in these computing models is that the CMP system has only one physical memory device per chip, while the traditional MP system has a one to one mapping between physical memory controllers and processor cores. This difference in the interface to the off chip memory leads to significant micro architectural challenges in the design of CMP memory systems. At a minimum, the bandwidth demands per memory block increase relative to the number of cores on the chip. Additionally, new layers of cache coherency are required to ensure that all inter-core memory references provide correct data. In addition to meeting the performance expectations of the multiprocessor computing domain, CMPs are becoming the standard processor architecture for the desktop market. As a result, they must also deliver compelling performance gains for the serial multiprogrammed application domain. Chapter 3 and Chapter 2 introduced performance improving techniques for the multi-programmed computing domain, which when combined with the cache system demands of multi-process workloads can be used to build a complete scalable CMP cache hierarchy system. This chapter introduces cache system design optimizations that leverages the characteristics of the different workload types in order to provide a unified performance model.

Chip designers continually leverage the additional transistor densities in each new processor generation to increase the core counts per chip. In doing so, the performance of the individual cores becomes less significant relative to the performance of the memory subsystem. The bandwidth pressure introduced by each core forces designers to spend large amounts of the transistor budget on last level cache storage. Due to the performance problems that are unique to CMP cache subsystems, it follows that the common practice of simply extending uniprocessor cache hierarchies is insufficient for large scale CMP systems. Additionally, the broad spectrum of application workloads that are

expected to achieve exceptional performance levels requires a calculated approach to cache hierarchy design. Optimization techniques that benefit parallel workloads often expose serial applications to significant performance risks. Alternatively, techniques to improve multi-programmed cache performance typically come at the expense of parallel application performance. There are two divisions of thought in current Chip Multiprocessor (CMPs) design: shared versus private last level cache (LLC) hierarchies. Neither model is sufficient for the requirements of both serial and parallel workloads. This chapter proposes a hybrid cache hierarchy design that leads to a net performance gain in both workload domains.

This chapter examines the CMP cache hierarchy from the perspective of offering support for both serial and parallel applications and provides solutions to the fundamental performance considerations facing CMP cache designers: **latency**, **capacity**, and **interference**. The proposed solution is a scalable last level cache architecture that offers access latencies comparable to the private cache architecture along with a relatively large capacity and reduced memory bus coherency traffic. Additionally, the proposed cache design protects the chip from applications that can damage the system performance with destructive cache interference. The proposed cache architecture is scalable in that it can be replicated easily with increasing core counts. Lastly, due to the scalability property, the last level cache architecture can support commonly used optimizations such as variable latency NUCA [40] caches and partitioning algorithms [24].

Shared caches provide high capacity low bandwidth systems. The Intel Core Duo [37] employs a shared last level cache in order to offer the maximum possible capacity per core for independent serial applications and to provide sufficient storage for parallel applications with shared data address spaces. Although these caches can offer impressive improvements in cache hit ratios, the design choice comes with the disadvantage of higher access latencies and greater exposure to interference from other co-scheduled applications. The alternative to the shared LLC is a private LLC. The AMD Athlon [2], deploys an array of private last level caches equal in number to the on-chip processor cores. This design improves the latency problem, however, it comes at the expense of a lower per core effective capacity and an increase in memory bandwidth. In the private LLC configuration, serial applications are permitted to access only one slice of the entire last level capacity. For parallel applications, private LLC designs incur an increase in replicated shared data across the cores. The amount of replicated data interference is a function of the degree of data sharing per application.

## 4.2 Workloads

### 4.2.1 Parallel Workloads

The applications studied in this chapter were programmed using fine grained software parallelism. In this model, there is one process that spawns software threads at specific points which are used to perform work on a shared data structure. The software threads can be applied to individual functions or even specific code regions within a function. The data address space associated with the heap is shared across the threads while each thread has private local stack space that is used to manage thread local variables. The instruction address space is also shared across the threads, keeping the instruction cache footprint small. All the parallel applications used in this thesis rely on the OpenMP thread model which is a software programming environment that facilitates the task of developing parallel codes. Section 4.2.1.1 describes the OpenMP programming model in detail.

#### 4.2.1.1 Open MP

Although the software thread model allows for fine grained control of parallel algorithms, it is difficult to use correctly in practice. Additionally, most existing software applications were written for uniprocessors and as a result are serial in nature. With the emergence of CMP processors in the market place there is a compelling need to convert these applications to a parallel programming model in order to exploit the benefits offered by CMP machines. The complexity of converting serial applications to a parallel software thread model means that there will be a significant time lag before the software industry can take full advantage of the performance offerings provided by CMP machines. Open MP [18, 60], however, is a multiprocessing paradigm that is expected to ease this transition. OpenMP hides the details of parallel programming in the compiler, while providing a simple and powerful programmer interface to the thread model. By simply inserting compiler directives around code regions that can be made parallel, the compiler generates software thread management code that handles synchronization and data sharing. This parallel programming model is making it considerably easier to convert serial applications to parallel.

Parallel applications running on CMP systems place considerably different performance requirements on the memory subsystem than their serial counterparts. The most fundamental performance decisions affecting parallel applications are **bandwidth** demands, cache access **latency**, last level storage **capacity** and the degree of **data sharing**. Programs with high levels of data sharing stand to benefit the most from large shared LLCs, however, this design strategy leads to large increases in LLC access latencies in addition to introducing the potential for interprocess interference in the serial

application domain. The parallel workloads used in this thesis are shown in Table 4.1. They are a mix of data mining applications from the NU-Mine Benchmark suite [22] and scientific workloads taken from the SpecOMP [17] benchmark suite.

Benchmark	64B Line Addresses	Working Set Size
NU-Mine Suite		
apr	-	-
genenet	-	-
hop	-	-
rsearch	-	-
semphy	-	-
snp	-	-
svm	-	-

Table 4.1: OpenMp Application Working Set Size

#### 4.2.1.2 Serial Workloads

When running independent serial applications on a CMP system such as the case with the Spec\_rate[17] benchmark harness, the applications benefit from the large effective capacity offered by a single shared cache. In this case, each core runs an independent workload making it possible for any one of the applications to exploit the additional capacity offered by the shared cache. For the same workloads running on a private cache organization, each of the workloads is limited to the maximum capacity provided by one cache slice. For applications with large working set sizes this is a considerable performance limiter. However, if the working set size is compatible with a single cache slice, then the private cache of Figure 1.1(b) is a higher performing alternative. In addition to having an equivalent hit rate of the larger shared cache in this case, the private cache has a faster access latency, making the average memory system latency decrease. A further benefit to the private configuration in this case is that each core is isolated from cache interference coming from the other cores. In this model, it is not possible for an application to acquire additional cache storage outside of its designated local cache. The trade off between interference and capacity is a significant challenge in the serial application domain, but recent trends in the computing industry seem to indicate a drive towards shared caches [37, 72].

In spite of the capacity benefits offered by the shared cache architecture of Figure 1.1(a), it has a significant design flaw, exposure to interference, that can lead to unpredictable runtime performance behavior. Chapter 3 introduced techniques to manage the allocation of storage resources across a shared cache for multiprogrammed serial applications. The replacement algorithms in uniprocessors are very effective at managing the allocation requirements of each application, however, simply reusing

these strategies in multi-core systems is sub optimal.

Figure 1.1(b) shows the private LLC organization. In this configuration, the total LLC capacity is equivalent to the shared case, however, the effective capacity is less because shared lines must be duplicated across the caches. In this model though, each private cache has a faster access latency than the shared cache making it possible for some applications to achieve greater performance due to the reduced memory latency. The working set size relative to the private cache size plays a significant role in determining the round trip latency. If the private LLC has sufficient capacity to cover the working set, then the private cache organization can outperform the shared cache. Coherency is maintained at the memory bus interface, which increases the off-chip bus bandwidth.

Chapter 3 introduced shared cache management techniques for the multi-programmed domain, and concluded that the techniques are sufficient for managing CMPs with low software thread counts. The burden imposed on the cache management system by large thread counts significantly increases the difficulty in identifying and correcting sources of interference. Based on this observation it is impractical to build, for example, an 8 core CMP with a shared last level cache running a cache partitioning algorithm. Rather, a more suitable approach is to restrict the partitioning algorithms to the configurations for which they perform the best, 2 way shared systems. In the serial application domain, the ideal last level cache would have interference management hardware built into a low access latency high capacity LLC. The combination of the need for scalable interference management policies and short latency large capacity last level caches leads cache hierarchy design in the direction of hybrid cache architectures.

### 4.3 Scalable LLC

The scalable cache hierarchy of Figure 4.1 is introduced as the design point where the performance demands of both serial and parallel applications converge. In this model, each core has a private first level data cache backed up by an LLC that is shared across two cores. The figure shows a 4 core CMP configuration, which in this case provides two LLCs. Coherency is managed at the bus connecting the LLCs to memory. In this model, there is only one peer LLC to snoop. This design leads to lower coherency traffic than with the private caches and as a result larger effective capacity. The increase in capacity occurs because fewer shared cache lines get replicated across the caches. Scaling this design to larger core counts requires the addition of one more LLC for every two cores. Increasing the LLC count as the core count increases ensures that serial applications remain isolated from the rest of the system while continuing to provide a large effective capacity per core. As the core counts continue to increase, a more fundamental CMP design question comes into play.

The bandwidth demands of very wide CMP systems may in fact require an increase in the physical memory controllers backing each group of processors. As a result, scalability may dictate that large CMP systems become MP systems integrated on a single chip. In this case, each cluster of cores would map to a unique physical address space via an independent memory controller. This design model will decrease the bandwidth demands on each of the memory controllers, but it will require more of the off-chip bus logic be integrated onto the chip. This is an area dedicated to future work since it extends beyond the immediate design space of the on-chip cache memory.

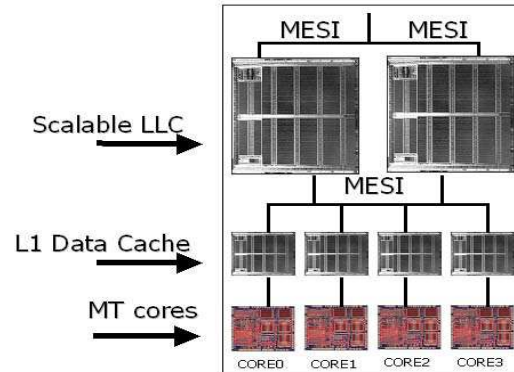


Figure 4.1: Scalable CMP Cache Hierarchy

#### 4.4 Parallel Application Memory Latency and Bandwidth

Capacity in MB	Tag Access Latency Cycles	Data Access Latency Cycles	Total
1	4	6	10
2	10	14	24
4	26	33	59
8	62	78	140

Table 4.2: Unloaded Cache Access Latencies

The figure below shows the capacity to access latency relationship over a range of LLC configurations. Each graph represents the average memory latency for three different cache organizations. The private cache model uses 4 independent 1MB caches, the scalable model has two 2MB caches and the shared cache is a single 4 MB cache. All configurations are 8 way associative and the total LLC capacity across the models is 8MB. The latency and capacity values used for each configuration are shown in Table 4.2 and were taken from [16]. Due to the relation between the unloaded cache access latency and the capacity, large caches must deliver a significant improvement in hit rate in order to overcome the latency penalty.

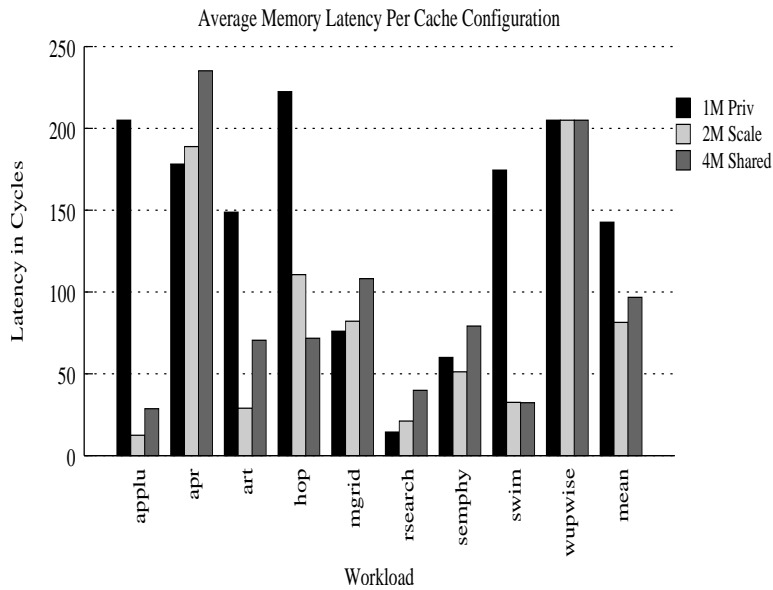


Figure 4.2: Average Memory Latency Across LLC Organizations

The average memory latency through the memory model is a strong indicator for the performance of the whole system. Figure 4.2 shows the average memory latency for each of the workloads in a 4 way CMP system across the range of cache configurations with a total capacity of 4MB. There are several interesting trends in the data. Workloads such as **apr**, **mgrid**, and **rsearch** show an increase in latency from the private configuration through to the shared LLC. This indicates that the working sets of each thread fit adequately in the private 1MB cache and that shared data hits are a relatively small component of the cache footprint. In this case the improvements in hit rate are small relative to the increase in latency of the larger last level caches. Figure 4.6 shows the composition of LLC references across the models in more detail. The next trend is covered by the workloads **hop** and **swim**, in this case the latency decreases across the configurations starting with the private cache down to the shared cache configuration. This indicates that the private LLC is inadequately sized for the workload. In the scalable model there is a clear increase in the overall hit rate that is large enough to offset the increased latency cost. Then, when the LLC jumps to 4MB there is another significant step in latency reduction that is attributed to a large improvement from effective capacity. Lastly, the more dominant trend shown by **applu**, **art**, and **semphy** is a drop in latency from the private configuration to the scalable followed by an increase in latency for the shared case. This trend reveals that the additional effective capacity acquired by the scalable caches is sufficient to satisfy the capacity demands of the workloads. The 1MB capacity per core in this case is not sufficient to store the working sets of many of the software threads. The capacity problem is compounded by applications that request remote data that must be replicated across the caches. The shared cache configuration has either an equal

or greater hit rate than the the scalable, but the LLC access latency in this case is large enough that it dominates the cache performance. In fact, as long as the scalable cache configuration round trip memory latency approximates the shared LLC, the hybrid model satisfies the performance demands of a unified cache hierarchy design metric. On average, across the parallel workloads the scalable cache hierarchy improves the memory latency of the private configuration by 43% and the shared configuration by 15%.

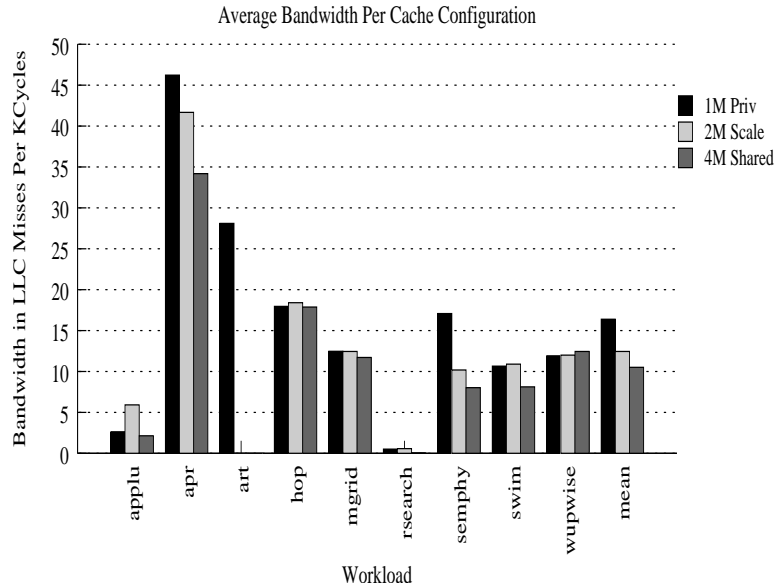


Figure 4.3: Memory Bandwidth Parallel Applications Across LLC Organizations

Managing the bandwidth demands of a CMP system are one of the more important roles of large last level caches. In this model, a large core count can cause a memory bandwidth requirement that is greater than the off chip bus can deliver. In this scenario the system performance is subjected to large unpredictable queuing delays. Scalable CMP systems must either control the bandwidth through large last level caches, or by increasing the number of memory controllers per core. In the latter case the chip would resemble a multiprocessor where each processor is itself a CMP composed of a small number of cores connected to an independent physical memory unit. Figure 4.3 shows the loaded memory bus bandwidth demands of the 4 way CMP system across the different cache configurations. In most cases the bandwidth makes a significant drop as the the cache configuration moves from private towards shared. The workloads **semphy** and **apr** exhibit the expected trend of a sharp decrease in bandwidth moving from the private to the scalable configuration followed by another drop going from the scalable to the shared model. The workload **swim** has virtually the same bandwidth in the scalable case as in the shared indicating that the shared cache lines are distributed uniformly across the cores rather than being concentrated in a pair of adjacent caches. The workload **applu** experiences an increase in

bandwidth relative to the private case which after cross referencing with the access composition graph of Figure 4.6 appears to be due to the large shift in hit rate across the cache configurations. In this case, LLC misses are being replaced by remote hits. On average across the workloads, the bandwidth of the scalable LLC is closer to the LLC configuration than it is to the private. It is 23% less than for the private configuration and 15% more than the shared. The bandwidth results further support the path towards hybrid LLC designs for CMP systems.

## 4.5 Serial Application Latency and Bandwidth

Workload Code	Component Benchmarks
ar.mc.eq.gp	art mcf earthquake gap
bz.tg.wp.ap	bzip2 tiger wupwize applu
ec.sm.mc.sw	eclat semphy mcf swim
gz.sm.mc.mm	gzip semphy mcf mummer
mm.tg.ec.sm	mummer tiger eclat semphy
sm.mm.ec.mc	semphy mummer eclat mcf
sm.wp.ap.eq	semphy wupwize applu earthquake
sw.ap.ar.bz	swim applu art bzip
tg.wp.sw.gz	tiger wupwize swim gzip

Table 4.3: Serial Workload Name Key

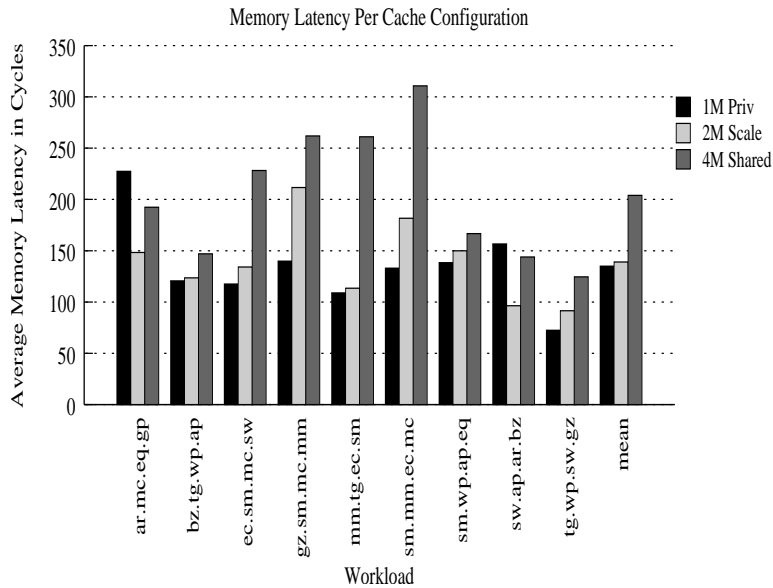


Figure 4.4: Memory Latency of Serial Applications Across LLC Organizations

The average memory latency for the serial applications was also measured. In this case the latency result reflects a system level or **capitalist** view of performance [32] since there is no mention of each application's individual performance. These measurements include the effects of interprocess

interference. In this case even though the large shared cache may offer the potential for an improved hit rate, it can be offset by the misses that result from inter-process interference. From Figure 4.4 it is difficult to isolate the components that contribute the most to the average latency. However, the three most dominant components are: unloaded access time, effective capacity, and interference. Figure 4.4 shows the latency for workloads consisting of 4 independent serial applications. There is no MESI protocol because the address space is private on a per core basis. The key used to translate the names of the workloads is shown in Table 4.3. The workloads **ar.mc.eq.gp** and **sw.ap.ar.bz** exhibit the desired behavior where the scalable hierarchy has the lowest latency of all the configurations. In this case the additional shared capacity available to the scalable hierarchy is large enough to offer a large improvement in hit rate without incurring a high interference penalty. The capacity gains from the scalable to the shared model are not large enough to balance the access latency. The majority of the workloads show a significant increase in memory latency for the 4M shared configuration. In this case, the effects of interference dominate the system performance and cause the majority of LLC references to miss. The comparatively lower latencies of the scalable cache for the workloads: **ec.sm.mc.sw**, **mm.tg.ec.sm** and **sm.mm.ec.mc** show the performance benefit of providing shielding from rogue applications in the system. In this case the effective capacity available to each application is only 2MB compared to 4MB in the shared case, but in this model any job can only interfere with one other job as opposed to 3 others in the shared model. On average, the scalable cache hierarchy offers a round trip latency within 2% of the private cache configuration and 30% less than the shared configuration. The applications used for these workloads were selected because they have the highest LLC miss components from their respective benchmark suites.

The bandwidth for the serial applications is shown in Figure 4.5. In this figure the trend is to decrease as the local LLC capacity increases. For the serial applications, moving from a private to a scalable cache covers many of capacity misses. Additional bandwidth is introduced due to interference misses, but the rate of interference misses is dependent on the relative LLC request rates for each application sharing the cache. In this case any bandwidth increases due to interference misses are insignificant relative to the private cache configuration. The same is true for the shared cache configuration; capacity limited applications have more storage available to them and can then shift the bus bandwidth demand to other applications. In this figure, the scalable bandwidth is 30% less than for the private case.

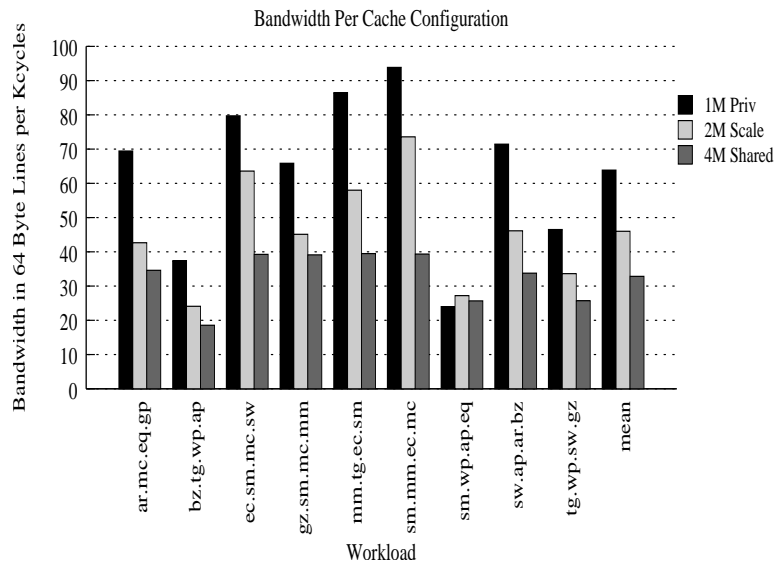


Figure 4.5: Memory Bandwidth of Serial Applications Across LLC Organizations

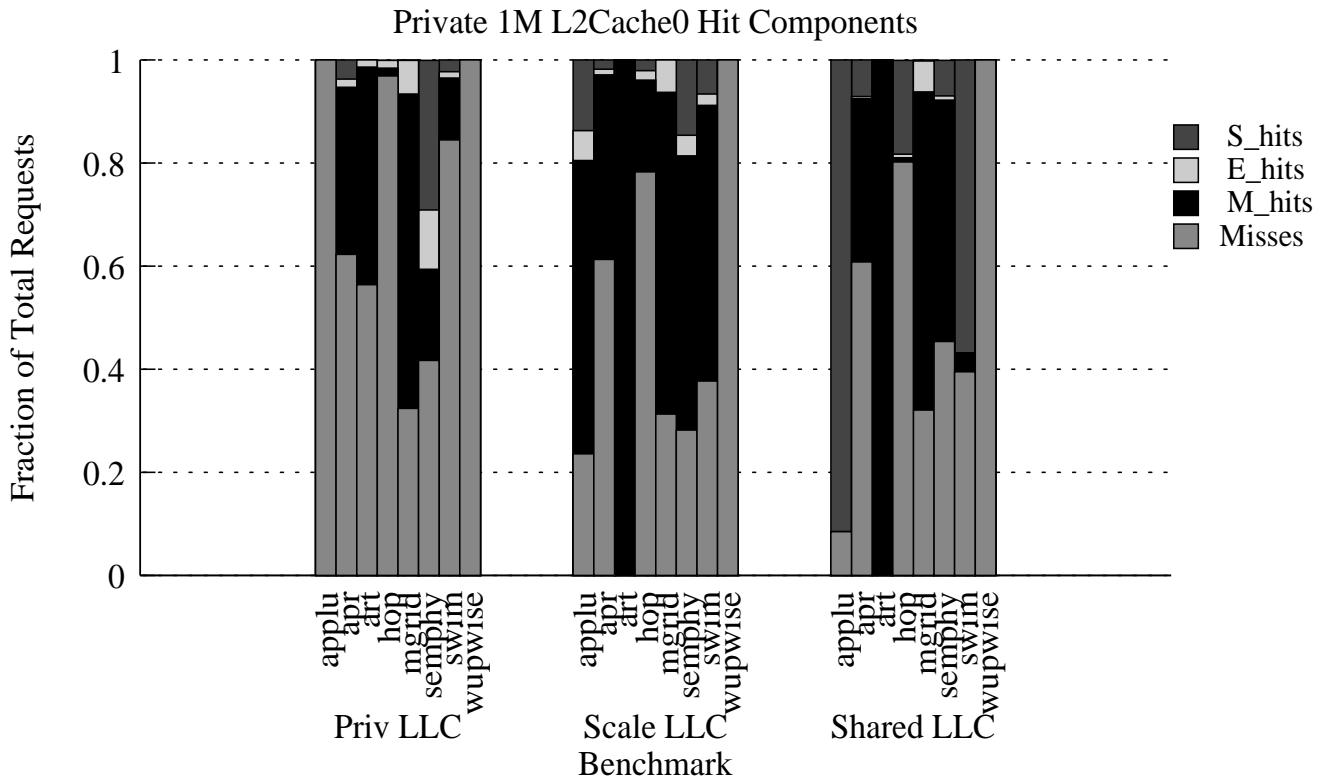


Figure 4.6: L2 Access Components

#### 4.6 Workload Analysis

#### 4.7 Parallel Workloads

To better understand the performance results from Section 4.4 the workload behavior is examined relative to the cache hit rates and the potential for interference initiated by sharing and inter-process

interference. Figure 4.6 shows the composition of LLC accesses for each of the cache models evaluated in this paper. One of the first workloads of interest is **art** OpenMP. In this example, the 1M private cache size has an average miss rate of 58%, however, when the scalable cache is used the increase in local storage leads to a 100% local hit rate. This shows that the cache capacity threshold for **art** is slightly more than 1MB. Once the individual threads are exposed to more local capacity the hit rate improves dramatically. This result also shows that the inter-thread sharing is limited to pairs of adjacent threads rather than a distribution across the whole pool. The OpenMP workload **applu** experiences two large steps in terms of the miss rate across the cache models. The extra capacity provided by the scalable LLC leads to a 75% increase in the hit rate, then it increases by an additional 10 percentage points as the capacity is increased to 4MB. From the results presented in Figure 4.2 the increased hit rate at this step is not sufficient to improve on the average memory latency, due to the large unloaded memory latency of the shared cache. The **semphy** workload has a large component of shared hits in the private configuration that lead to a large drop in the miss rate (23%) for the scalable configuration. This improvement in miss rate translates into a significant round trip latency time as shown in Figure 4.2. For the shared cache model running **semphy**, the miss rate is slightly higher than in the scalable configuration. This is an artifact of the different latencies used for the simulations. In this case, the lower latency used in the scalable configuration means the workload advanced into a different program phase that was not reached by the shared case.

#### 4.7.1 Serial Workloads

One of the primary goals of this thesis is to define a cache hierarchy design that exploits the performance attributes of both parallel and serial workloads. In the serial workload domain, one of the most significant design flaws with a normal shared cache is that there is no way to manage the cache resources on a per application basis. The problem with this model is that any application can commandeer the cache storage at the expense of all the others regardless of the priority of the other applications. This is referred to as inter-process cache interference. The challenges in defining and measuring the shared cache performance in this case are described in [32]. The fact that one application can change the runtime behavior of any application on the system is especially problematic for embedded systems where hard real time scheduling deadlines dominate the application space. The following section examines the interference component for the serial workloads shown in Table 4.3.

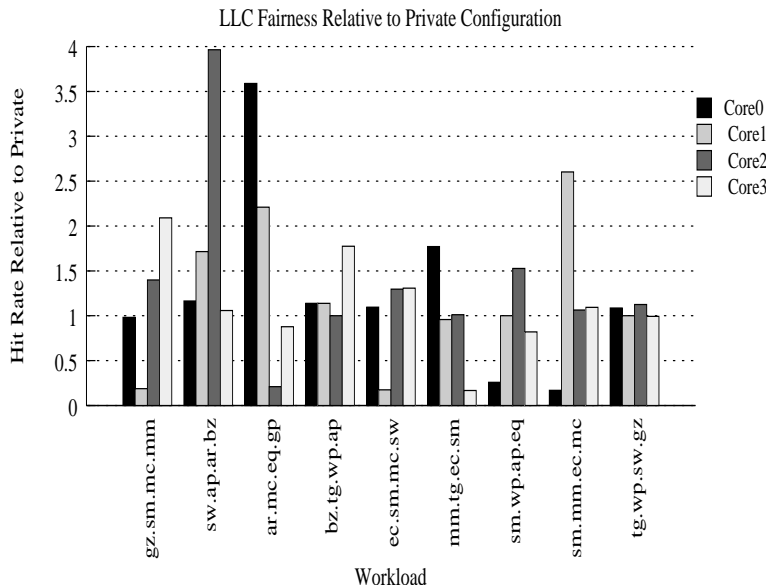


Figure 4.7: 4M Shared LLC Cache Fairness

#### 4.7.2 Fairness

Figure 4.7 shows the performance to fairness relation for a shared 4MB LLC relative to the baseline 1MB private cache. In the shared cache model, each core has an average of 1MB of storage, but because the 4MB is shared the applications have the potential to leverage this additional storage. The fairness is derived from the change in hit rate that each application sees when co-scheduled on the shared cache with 3 other applications, against the hit rate when the same application is run in isolation on a 1MB cache. This relation is shown in Equation 4.1 where the hit rate of the application in multiprogrammed mode is the numerator and the uniprocessor run is the denominator.

$$\delta hit = \frac{HR_{dual}}{HR_{uni}} \quad (4.1)$$

A value of 1 for any workload indicates that the hit rate was unchanged from the private to the shared cache mode. Values greater than 1 indicate an increase in hit rate which is due to the increased effective capacity, while a decrease represents a drop in hit rate that is attributable to interference. The figure shows a wide range of fairness values. The significant divergence from the normalization point illustrates the performance dependence that each job has on the others. The purpose of this figure is primarily to highlight the problem of interference that is inherent in shared caches. Specifically, the greater the number of applications that can share the cache, the greater the variability in runtime performance per application.

The scalable cache architecture address the problem of interference to a first order approxima-

tion. By limiting the number of process that share a cache to two, the scalable architecture protects the rest of the system from interference. By selecting the scalable architecture, cache partitioning algorithms optimized for low thread counts such as those introduced in Chapter 3 and from prior studies [24, 13, 42, 61] can be implemented in each LLC slice. Figure 4.8 shows the fairness results from a scalable cache where each cache is 2MB making the total effective capacity 4MB. This experiment did not run a partitioning algorithm, but the per slice LLC results can be compared to the partitioning algorithms run on the same applications in Chapter 3. The figure is similar to Figure 4.7, with the basic exception that the extrema away from the normalization point are muted. This reinforces the design goal of building scalable technologies that can reuse existing optimizations based on smaller less complex systems.

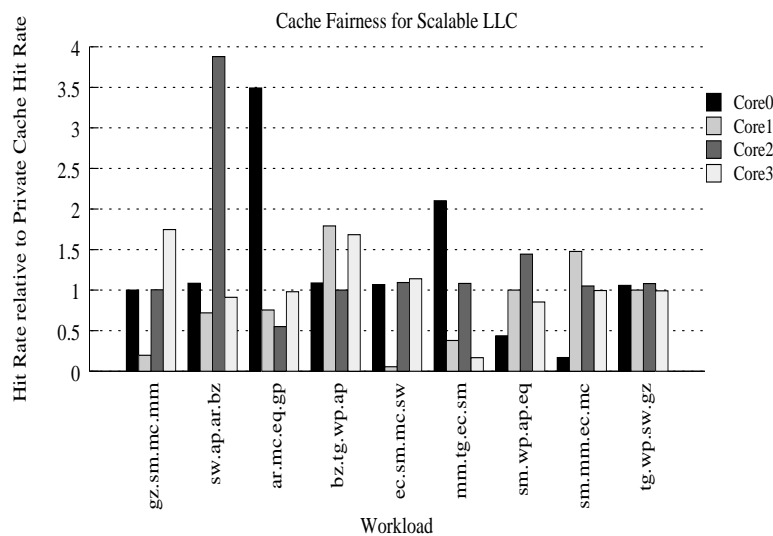


Figure 4.8: 4M Scalable LLC Cache Fairness

## 4.8 Summary

This chapter introduced the workload characteristics of parallel applications and showed the related impacts on cache performance. The effects of latency, bandwidth, and cache capacity were examined relative to the parallel workloads over the shared and private cache design space. The scalable cache LLC based cache hierarchy was introduced as a low complexity highly effective approach to designing memory subsystems that deliver high performance in the parallel and serial application domains. The issue of off-chip memory bandwidth was identified as being a potential limiter to the upper bound of CMP core count scaling.

## Chapter 5

### Experimental Method

#### 5.1 Simulation Infrastructure

The cache hierarchy experiments were performed using a memory subsystem simulator driven by the PIN [58] binary instrumentation tool. The simulator supports both parallel and serial applications. For the serial case, the it invokes PIN once for each application binary that is run on the system. A PIN tool was developed to emit a trace of the virtual addresses for all data memory references. This trace is sent through a UNIX pipe where it is consumed on the fly by the cycle accurate memory simulator. The named pipe performs flow control between the benchmark applications and the cache simulator. When the pipe is full, the application stalls until data is consumed by the simulator. Applications are run until either a predefined number of memory references have been issued, or the end of the program is reached. When more than one application is being simulated, the simulation ends after all the jobs have reached their respective termination conditions. For the parallel workloads, the PIN tool is run in multi-threaded mode so it can capture the trace of all the software threads. In this mode, pin is invoked only once per application and it produces one trace for each software thread. The memory simulator consumes data from all of the traces and issues memory requests to the respective first level caches. The memory simulator provides a fully configurable detailed storage and bus timing model. The microarchitectural details of the simulator are shown in Table 5.1.

Module	Size	Latency
Core Count	4	-
Data Cache	16KB 4wy	2 cycle hit
Instruction Cache	NA	NA
L2 Cache	4MB 8wy	varied from 10 - 59
Memory	4GB	200 cycle
Memory Bandwidth	16 64Byte Lines per cycle	

Table 5.1: Memory Simulator Configuration

In order to provide support for parallel applications, the memory simulator includes an implementation of the MESI cache coherency protocol [45]. Since parallel applications rely on software contexts that share a common address space, a single cache line can be referenced by any number of software threads. In serial applications however, each software context has exclusive ownership of the data in any cache lines that it references. To address this issue, the MESI protocol enables the cache hierarchy to determine dynamically which cache lines are shared. When parallel applications are considered, sharing refers to individual cache lines rather than cache sets. This raises an important design consideration for shared CMP caches because in addition to preventing individual applications from excessive consumption of cache storage resources, there still needs to be sufficient common storage to exploit the benefits offered by shared memory, parallel applications.

The cache hierarchies introduced in Chapter 4 are modeled with a combined LLC capacity of 4MB, this was chosen because it was large enough to completely fit some of the working sets, while causing other workloads to be capacity constrained. In some cases the cache size was varied to understand the effects of LLC capacity on the memory performance. In these cases the LLC capacity ranged from 1MB to 32MB of total storage. For the serial workload case, the applications are fast forwarded for 200M data memory references each, followed by a 100M reference warm up interval. Statistics are collected for 300M memory references after the warm up phase completes. For the OpenMP workloads the fast forward interval lasts until the main thread reaches the spawn point for the other software threads. The caches are warmed up for 100M references, after which the main simulation runs for 500M total memory references.

## 5.2 Workloads Studied

Experimental workloads were chosen based on the relation between their working set sizes and their LRU stack profile. This thesis concentrates primarily on memory intensive applications. For a CMP processor, the goal is to enable all applications to make forward progress as close as possible to their isolated performance. This is a fairness issue that is more important in multi-core systems than multithreaded processors because each core is expected to achieve compelling performance. The fairness criteria means that the cache system should be resilient to rogue applications that attempt to take over all of the cache storage resources. The benchmark applications were all compiled with gcc at optimization level O3 and run on Intel Xeon processors.

The workloads used in this study were pairs of benchmarks chosen from Spec 2000 integer and floating point, Spec OpenMP, NU-MineBench, and BioBench benchmarks. A mix of serial and parallel applications were studied in order to explore the design space of a unifying CMP cache hierarchy. Ta-

Benchmark	Line Addresses	Working Set Size
NU-Mine Suite		
bayesian	2311	147904
eclat	303626	19432064
semphy	2937941	188028224
utilitymine	88015	5632960
Bio Informatics Suite		
clustalw	28905	1849920
hmmer	183716	11757824
mummer	2795313	178900032
protdist	23739	1519296
protpars	193482	12382848
tiger	2147723	137454272
SpecFP 2000		
168.wupwise	1967305	125907520
171.swim	3121262	199760768
173.applu	2856295	182802880
179.art	55687	3563968
183.equake	679582	43493248
SpecINT 2000		
164.gzip	71282	4562048
176.gcc	125464	8029696
177.mesa	110093	7045952
181.mcf	171620	10983680
186.crafty	31629	2024256
197.parser	207330	13269120
253.perlbnk	401046	25666944
254.gap	3148746	201519744
255.vortex	400251	25616064
256.bzip2	245636	15720704
300.twolf	20670	1322880

Table 5.2: Working Set Sizes

Table 5.2 shows the working set sizes of the benchmarks used in this thesis. On average, the floating point benchmarks have much larger working sets than the integer, making them more bandwidth intensive and more prone to causing inter-process interference. NU-MineBench is a collection of data mining applications that represent several classifications of data mining algorithms, clustering, classification, association rule discovery, deviation detection and specialized mining. They were programmed with the OpenMP thread model, making use of fine grained software parallelism. The SpecOMP suite is a collection of the Spec2000 FP benchmarks that were reprogrammed under the OpenMP software model. The BioBench suite represents a set of algorithms from the bioinformatics application domain.

### 5.3 Real Hardware Measurements

### 5.4 Performance Monitors

In addition to modeling the memory subsystem, the workload performance of a real multi-threaded machine was measured to provide motivational data for shared cache management optimizations (Chapter 2). These experiments, were conducted on an Intel Xeon, 2 way hyper-threading

processor. Pairs of applications were co-scheduled and various metrics were measured by sampling the associated performance counter registers. The performance counters were read with the user level API supplied by the **perfctr** [11] Linux kernel module. The base Enterprise Linux kernel v 2.6.9-22 was patched to provide the **perfctr** module. Pairs of memory intensive serial applications were co-scheduled on the Xeon processor and the performance counters relative to each hardware thread were sampled and recorded.

## 5.5 OS Scheduler Modifications

The Linux scheduler from kernel version 2.6.0 was modified to support the activity vector based scheduling algorithm of Section 2.2.2 and was run on an Intel Pentium 4 Xeon processor with hyper-threading enabled. This processor has two contexts, or virtual CPUs, that share the cache hierarchy, data registers, and functional units. The default Linux scheduler runs a symmetric multiprocessing (SMP) scheduling algorithm on each virtual processor to select the next job to run. The algorithm was adjusted to include activity vector information in the job selection criteria. Rather than treat the activity vector information as a score in the scheduling priority algorithm, the default scheduler is first used to select the best job to run. Then, it queries the activity vector scheduler to search for a job that is expected to have fewer resource conflicts with the job running on the other virtual processor. If the activity scheduler can not identify a better job, then the default job is allowed to run. Since the Pentium processor used in these experiments can not be modified, activity vector support was modeled in software. The details of which are described below.

In order to expose cache activity information to the operating system each benchmark application is profiled, during which time the activity vectors are generated. The Intel Pentium 4 Xeon processor does not provide a mechanism for collecting cache miss statistics on a per line granularity. Thus, in order to approximate the actual memory behavior of a given job, each benchmark application is run under the Valgrind memory simulator. Valgrind provides a model of the actual cache hierarchy of the host machine along with a record of runtime program information, such as the instruction count and cache miss rate. This simulation stage is used to record all memory requests and misses associated with a given cache set. The access and miss counts for each set are recorded in software activity counters every program phase interval of 50 million instructions. As described in Section 2.2.2, a bit is set in the activity vector if the corresponding counter exceeds a threshold value. The resulting activity vectors for each cache level are written to a text file in ASCII format. These vectors are later passed to the operating system kernel memory space, where they are referenced by the job scheduling algorithm on the second phase of the scheduling optimization. The results of the activity vector experiments are

presented in Chapter 5.6

## 5.6 Linux scheduler implementation

### 5.6.0.1 Algorithm details

---

**Algorithm 2** Modified linux schedule() function.

---

```

Require: current_active_task
Ensure: next_task_ready
if need_resched then
  log_task_runtime()
  th_sample_cache_vector(task)
  update_context_switch_count()
end if
if pick_next_task then
  if runqueues_not_balanced then
    load_balance()
  end if
end if
next_task = select_highest_priority_task()
next_task = th_schedule_cache_vector(next_task,
  other_cpu_runq, task_priority_list)
recalc_task_prio(next, timestamp)
if switch_tasks then
  unmodified linux source
  ...
end if

```

---

Algorithm 2 is a pseudo code representation of the activity based scheduling module and its interface to the kernel scheduling software. The scheduling module is a tool that provides a user level interface to the kernel scheduler, which can be used for both collection of statistics, and the activation of the scheduling algorithms discussed in this paper. The module was designed so that it could be recompiled and run without rebuilding the Linux kernel. To achieve this goal, the module exports 2 functions to the Linux schedule() function. At runtime, the scheduler queries these functions in order to select which job to activate at each schedule invocation. The first function th\_sample\_cache\_vector(), shown in Algorithm 3, is used to sample hardware performance counters and record their associated data for each of the jobs running on the system. The second function, th\_schedule\_cache\_vector(), shown in Algorithm 4, is used to select the next job to run based upon analysis of the cache activity vectors.

The schedule() function of Algorithm 2 has two important code regions that were chosen as entry points for the activity based scheduling module. The first section is labeled ‘need\_resched’, here

---

**Algorithm 3** Performance monitoring function.

---

```

Require: current_running_task
Ensure: Global perf counter tables updated
  if logging_l2cache_perf then
    sample_perf_counter(L2CACHE, cpu_id)
    log_counter(bmark_name)
  end if
  sample_perf_counter(icount, cpu_id)
  store_icount(task_id)

```

---

the available tasks either get removed from the `run_queue` if they have completed, or their scheduling priority gets updated. The `th_sample_cache_vector()` function was inserted into this section to monitor the state of the cache system performance counter registers associated with the presently active tasks on each of the logical CPUs. The performance counter information was stored internally in the kernel memory space for use in both statistical reporting, and as a reference point for the scheduler optimization. The second important region in the function `schedule()` is named ‘`pick_next_task`’. Here, as the name suggests, the kernel selects the next job to activate on the CPU that the scheduler is currently running. The function `th_schedule_cache_vector()` was introduced here to select the best job to run based upon the activity vector information that was passed to kernel memory by the user.

#### 5.6.0.2 Kernel module details

Upon completion of the profiling stage for each of the benchmarks, the user level profiling tool is used to copy the contents of the vector files into the kernel memory. This tool is a device driver that enables a user to control when the scheduler should activate the `th_sample_cache_vector()` and `th_schedule_cache_vector()` functions. When the user interface is invoked, the default Linux scheduler is active. The user is prompted for the file names of the activity vector files, then each is read into kernel memory for later use in the scheduling algorithm. Once the vector files have been read in successfully, the user selects the desired scheduling algorithm, then starts all of the benchmark applications in the workload set. The total execution time for the workload set is recorded, at which point, the user can command the kernel to return to the default scheduling mode.

#### 5.6.0.3 Linux scheduler opportunity

Figure 5.1 illustrates the scheduling opportunities available to the Linux scheduler for a range of workloads. Each time the scheduler is invoked on one of the logical CPUs it has a pool of ready jobs to choose from. The scheduler was instrumented in order to support runtime analysis of the job pool. The data in Figure 5.1 show the ratio of the number of times the scheduler was invoked to the number

---

**Algorithm 4** Vector scheduling algorithm.

---

**Require:** *ready\_task*, *runq\_other\_cpu*,  
*active\_task\_other\_cpu*, *task\_priority\_array*

**Ensure:** *next\_active\_job*

*initialize\_job\_weights()*  
*get\_vectors(ready\_task, other\_task, icount)*

**for all** *Cache\_Levels* **do**  
    *res\_vec* = AND(*vector\_ready*, *vector\_other*)  
    *overlap\_bits* = count\_result\_bits(*res\_vec*)  
    update\_weight\_function(*overlap\_bits*)  
**end for**

**for all** *tasks\_in\_run\_queue* **do**  
    compare\_vectors(*task*, *task\_other\_cpu*)  
    **if** *weight* < *best\_weight* **then**  
        *best\_task* = *last\_job\_in\_queue*  
    **end if**  
**end for**

---

of times a benchmark could have been selected in order to reduce cache resource conflicts. Note that in order for the activity based scheduler to be invoked, the job running on the other virtual processor must belong to the experimental workload. On average, more than 45% of schedule invocations trigger the activity based scheduler. The remainder of schedule invocations are to handle operating system tasks that are independent of the workload set. This indicates that there is considerable headroom for the selection of better jobs for co-scheduling.

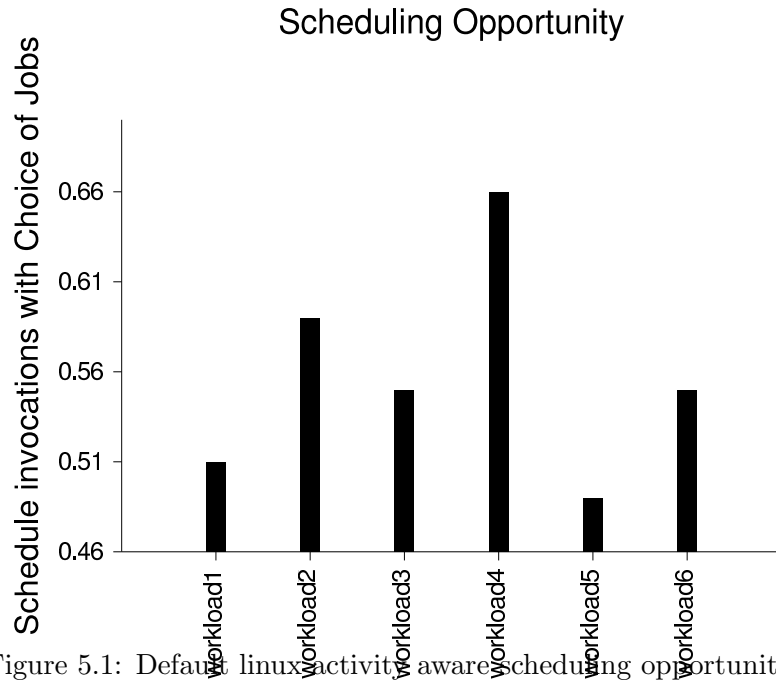


Figure 5.1: Default linux activity aware scheduling opportunity.

## Chapter 6

### Related Work

#### 6.1 CMP Cache Hierarchies

Shared CMP cache hierarchies have received considerable attention in the literature recently. The work has focused primarily on three topics, LLC access latency, memory bandwidth, and inter-process cache interference. The workloads run on the system generally determine the relative importance assigned to each of the above optimization points. Parallel workloads have led to much of the work in optimizing LLC access latencies, and in particular have led to the development of Non Uniform Cache Access (NUCA) memory subsystems [9, 16, 40]. Applications with shared address spaces are amenable to large shared last level caches due to a high degree of data sharing across the application threads. As a result, the optimization techniques have focused on providing low latency accesses to large cache structures. Keckler and Burger [40] introduced the concept of building large caches out of blocks of smaller low latency caches for uniprocessors. The goal in these models is to place the most frequently accessed data in a small low latency cache located closest to the core. This approach reduces the latency penalty attributed to the wire delays required to span the physical area of multi megabyte caches. The resultant model requires a placement algorithm to determine when and where to move cache blocks targeted for optimization. This work was expanded in [16] and [9] to account for the latency problem in CMP caches. In CMP caches the primary challenge is to build large shared last level caches with the equivalent latency of an array of private caches. To meet this design goal [16] introduced a technique to make the tag arrays private per core while building the data arrays in a large shared L2 cache structure. This approach allows the tag references to have the latencies of the private cache case while making the reference to the actual data require the same latency as the corresponding shared cache. To save capacity, the data arrays do not need to be duplicated following remote cache tag hits. Although this technique does an adequate job of lowering the shared cache latencies for parallel applications, it does not provide a mechanism to prevent applications from exposure to interference from others. Additionally, the private tag arrays limit the amount of tag

storage available to independent applications running on the system. Tiled cache hierarchies for CMP processors also fit in the domain of NUCA cache policies. In [77] high core count scalable CMPs are constructed from tiles of small CPUs. Each tile contains a core, first level I and D caches and one slice of a system wide last level cache. In this model the LLC slices are connected over a network. The network helps reduce the rate of increase of cache latencies relative to cache size. By building the LLC out of slices, the slice size determines the access latency coupled with the network hop time to reach the two most distant nodes. The rate of increase in network hop time is considerably less than in it is relative to cache capacity increases.

Other techniques have focused on managing the the shared cache resources to control the performance losses associated with inter process interferences. Hsu [32] focuses on the metrics used to measure performance of CMP machines. This paper makes the observation that the definition of performance is variable in CMP systems and performance optimizations depend on the definition. Shared cache performance can be a function of the fairness of storage resources allocated to each application, or it can be tied to the performance of the highest priority applications in the system. Regardless of the definition however, they conclude that process aware cache allocation algorithms are needed to provide peak performance on CMP memory systems. In [61] cache partitioning algorithms were introduced to provide dynamic management of shared cache resources in CMP systems. This work proposed dynamic feedback directed optimizations aimed at detecting and correcting inter-process cache interference in the cache hardware. This work built on earlier partition algorithms proposed in [24, 13, 42].

The last category, managing memory bandwidth was confronted in [47, 68, 1]. Both studies focused on techniques to limit the memory bandwidth requirements of CMP systems. Large shared last level caches again proved to offer the most benefit in reducing bandwidth, so both works focus on latency reducing techniques that enable large shared last level caches. In [47] the LLC was divided into smaller low access latency blocks that were assigned to specific processor cores based upon a programmable look up table. The technique addresses both latency and exposure to interference from co-scheduled applications. While the proposed shared cache was expected to provide low access latencies the capacity was compromised by the need to copy shared data across cache slices. The capacity decrease due to replication was not problematic in this study because the cache sizes investigated were considerably larger than the working sets run on them. In general though this approach was able to outperform the private cache case because the latencies were set low to reflect the slice sizes of the L2 cache. Additionally, the algorithms used to map the caches slices to cores are complex and will likely become more so as the application count increases. In [68] a large third level cache was the proposed

solution to the bandwidth problem. In this model, each pair of cores was connected to a second level cache. All the second level caches were backed up by a large L3 cache that could be resident either on or off chip. This particular L3 implementation was designed as a victim cache for the L2 caches so it shares the same bus as the L2 caches and physical memory. This study focused primarily on optimizing write back operations that would limit the pressure on the L3 cache. Although in this work the L3 cache was not part of the data path from memory to L2, it offered relatively short access latency for the reuse of cache lines that had been evicted from L2. Having a shared structure across multiple cores however introduces the problem of exposure to interference back into the system that the private L2 caches helped reduce.

## 6.2 Innovative Uniprocessor Cache Organizations

Numerous techniques for optimizing memory performance on uniprocessors have been proposed. The victim cache was first introduced by Jouppi in [39] as an inexpensive way to provide data cache performance without a significant size increase in the cache. More recently, power and performance trade offs of memory system optimizations were evaluated in [8]. This work used techniques such as victim caching as a means for providing good performance on a low power budget. The study was performed on a super scalar machine with only one level of on-chip cache.

Innovative caching schemes that exploit the variability of spatial and temporal locality in applications have emerged as efficient ways to retain important data in the highest cache levels [69, 25, 14]. Gonzalez et al. [25] proposed a partitioned data cache that could handle spatially and temporally local cache lines. This model enabled a general purpose processor to dynamically adapt to the runtime memory access footprint. For some vector based applications, the large arrays are accessed frequently meaning that several adjacent cache blocks need to be retained in the cache for future access. This behavior benefits from wide cache lines, while other control intensive applications would suffer from internal fragmentation if the cache lines were this size.

## 6.3 Cache Sharing

While there exists a large body of literature concerning cache performance for uniprocessors, there is comparatively little on the subject of caching for multithreaded processors. Cache partitioning was first introduced as a way to improve the performance of workloads running in a time-shared uniprocessor environment [70, 15]. The concept of column caching was introduced by Chiou in [15] as a way to make the timing of cache accesses for real-time embedded systems applications more predictable.

Recently, cache partitioning has been used in the domain of multithreaded processors to address the memory latency problems introduced by a cache shared between simultaneous jobs. In particular Suh et al. [71] introduced the idea of hardware cache partitioning to explicitly manage the allocation of cache storage for multithreaded workloads. This work introduced the marginal gains counters which were used to determine the best cache partition for the coming sampling interval. This work focused primarily on the storage capacity of each job and did not explicitly address the issue of inter-thread interference.

Kim et al [42] examined the cache partitioning problem from the point of view of thread fairness. They based the partitioning mechanism on the LRU managed scheme used in [71] and improved it by balancing the partition decisions with a set of fairness metrics. The principle drawback with this technique however, is that it requires profile information from each of the jobs in order that their isolated MPI rates are known at runtime. Additionally, this work focused on a system with only 2 threads.

The impact of inter-thread interferences between workloads of 2 jobs was studied by Chandra et al in [13]. In this work, they focused on predicting the number of cache evictions that would be introduced to a given thread if it shared the second level cache with another job running on a CMT processor. While this work reveals important insight into the nature of interferences between jobs it does so only for 2 jobs at a time and it does not specifically look at hardware techniques to exploit this information.

Other researches have since used column caching for different problems. Carzola et al [12] used it for partitioning the L2 cache in order to solve a QoS problem in SMT processors. Here the goal was to partition the cache so that one high priority application would receive very little interference from the other jobs in the system. While this technique does implement column caching, the goal of the work is not system performance and therefore requires a fundamentally different partition controller. Garcia et al examined the impact of cache sharing on multithreaded systems from the point of view of the data cache [23]. They examined static partitioning models but concluded that dynamic cache partitioning was needed to significantly improve the performance of caches for multithreaded processors. While there has been recent studies that propose various schemes for cache partitioning, the literature still lacks a body of work that explicitly addresses interferences between threads in a system of more than 2 threads, and applies a dynamic hardware solution that does not rely on prior knowledge of each job's execution characteristics.

## 6.4 OS Scheduling For MT Performance

In response to the importance of resource contention between threads associated with the cache hierarchy, there have been a number of operating system level solutions to solving this problem. Jack Lo et al. demonstrated in [48] that database applications which suffered from inter-thread conflict misses could be improved significantly by using the operating system to change the way virtual memory pages were mapped to their physical counterparts. This approach made it less likely that data belonging to different thread contexts would map to the same cache sets, thus reducing the amount of inter-thread conflict misses. Symbiotic job scheduling [67] was introduced by Snively and Tullsen as a method to improve performance by making the operating system co-schedule threads that were likely to ‘behave’ well with one another. While this technique exposed the potential headroom for using the operating system to advance SMT performance, it did not reflect closely enough the runtime environment of a real operating system job scheduler.

Parekh, Eggers, and Levy proposed exposing microarchitecture level performance information to the operating system job scheduler in [56]. This work showed IPC improvements on the order of 10%, in a simulated operating system environment. In [71], the overall cache demands of threads is predicted using fine grained cache monitoring. By tracking which cache lines and sets are most recently used and how often they are used, they can track overall cache demand for each thread. Threads are allocated a different number of sets within the cache based on predicted hit rates, and scheduled based on a greedy static assignment algorithm.

## 6.5 Transition to Multithreaded Processors

Multithreading processors are becoming the industry standard for high performance computer systems. Numerous thread models exist which emerged as solutions to specific performance issues. The initial implementations were designed to improve the performance of shared memory multi processor computer systems. Researchers observed that individual nodes were spending significant portions of their execution time idle waiting to service off-chip memory accesses.

### 6.5.1 Course Grain Multithreading

The APRIL processor from MIT was a course grain multithreading implementation of a single processor (CGMT) that could execute instructions from a different software context while the main context was servicing a memory request [3]. Gupta and Weber [76] were attracted to CGMT for similar reasons, they had identified the increasing memory latency due to remote memory accesses

in a multiprocessor system. They observed that the performance improvement due to multithreading was application dependent, and was also limited by the interference introduced by the addition of instructions from other hardware contexts. Other multi-processor systems such as the Data Diffusion Machine [53] relied on the coarse grain thread model to hide the latency associated with off-chip memory requests. Recently, IBM released the Power PC RS64 IV [10] which was the first commercial implementation of a coarse grain multithreading processor.

Lim and Bianchini quantified the performance limits of multithreaded processors in [46] by introducing a model used for predicting the performance gain of a workload with the introduction of multithreading. While the results indicated that multithreading could improve the machine throughput performance, they observed that the performance of these machines was limited by the context switch time and the number of hardware contexts provided. Similarly, they observed that performance could be lost if either of these parameters were too high. Muller, Stallard, and Warren[53] examined the scalability of multiprocessor systems using multithreaded processor nodes. They studied parallel applications running on a multi processor system and examined multithreading with different instances of the same process sharing an MT processor. In addition, they highlighted the problems associated with destructive cache interference.

### 6.5.2 Fine Grain Multithreading

The introduction of fine grained multithreaded processors preceded the widespread use of superscalar architectures. In [21] the fine grained thread model was motivated from the need to increase the utilization of processors with deep pipelines. Rather than perform control speculation and out of order execution to maintain a steady supply of independent instructions, this model proposed alternating each issue cycle with instructions from a different process. Interleaving the instructions from independent contexts, provided time for the processor to resolve register dependencies and other hazards associated with each program. Reducing these dependencies prevented pipeline stalls from occurring and led to an increase in processor utilization. Loikkanen explored a number of fine grained issue policies based on a superscalar processor core in [51] to improve the throughput of superscalar machines. Earlier fine grained thread models such as the HEP, DASH, MASA machines [66, 44, 27] relied on filling the pipeline with instructions from independent processes. This approach relied on having a large pool of jobs to choose from to keep the pipeline fully utilized.

### 6.5.3 SMT Simultaneous Multithreading

Simultaneous multithreading later emerged as a low cost means to improve the utilization of superscalar processors. Wide issue superscalar processors proved effective for exploiting instruction level parallelism, however control and data dependencies limited the number of independent instructions available in a given instruction sequence. By enabling the issue of instructions from one or more contexts each cycle, periods of low ILP could be compensated for by thread level parallelism (TLP) [73, 52, 19]. Simultaneous multithreading has since emerged as a popular platform for high performance processors. It has proved very useful for computing environments where several jobs are run in parallel. However, SMT has not addressed the problem of improving the performance of individual jobs.

### 6.5.4 Performance Issues

There have been numerous studies focused on improving the throughput performance of simultaneous multithreading processors. The techniques for solving this class of problems are covered by a range of disciplines, including: microarchitecture level solutions, operating system thread scheduling, and compiler level optimizations. Although this thread model has proved to improve the throughput performance of multi-job workloads, performance bottlenecks related to the shared hardware resources on these processors have placed limitations on their efficiency. In particular, the shared cache hierarchy [29], and the instruction fetch and issue logic [20] have emerged as components that require special attention in an SMT environment.

In [74], Tullsen et al. studied instruction fetch and issue techniques at the microarchitecture level that provided the best processor utilization for a set of jobs on an SMT processor. The ‘ican’ issue policy was introduced as a method for assigning fetch and issue priorities to the hardware contexts resident on the machine. Lo et al. examined additional techniques for maximizing instruction throughput in [50]. Hily and Seznec demonstrated in [30] that the performance of multithreaded systems was limited by the cache hierarchy, in particular, the increase in bus contention introduced by the additional memory requests in an SMT machine model. This observation inspired later studies that focused on improving the latency tolerance of SMT architectures,

### 6.5.5 Compilers

Studies of the impact of compiler optimizations on SMT performance [49] show specific cases where an SMT processor can benefit from changing the compiler optimization strategy. In particular,

they showed that cyclic iteration scheduling is more appropriate for an SMT processor because of its ability to reduce the TLB footprint. In addition, the work showed that software speculative execution can be detrimental to SMT systems, because it decreases useful instruction throughput. According to [49], although the latency-hiding benefits of software speculation may be needed less on an SMT, the additional instruction overhead introduced by incorrect speculation might severely degrade performance. Kuhmar and Tullsen investigated compile time methods for reducing the amount of interference between threads in the instruction cache in [43]. This work focuses primarily on the icache and is specifically aimed at static, rather than dynamic optimization techniques.

## Bibliography

- [1] B. Jacob A. Jaleel, M. Mattina. Last level cache (llc) performance of data mining workloads on a cmp a case study of parallel bioinformatics workloads. In The 12th International Symposium on High-Performance Computer Architecture, February 2006.
- [2] Advanced Micro Devices. Amd athlon 64x2 dual core product data sheet. Technical Report 33425, Advanced Micro Devices, Sunnyvale, CA, May 2005.
- [3] A. Agarwal, B. Lim, D. Kranz, and J. Kubiawicz. APRIL: A processor architecture for multiprocessing. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 104–114, Seattle, WA, 1990.
- [4] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In Proceedings of the 27th International Symposium on Computer Architecture, pages 248–259, May 2000.
- [5] A. Alameldeen and D. Wood. Adaptive cache compression for high-performance processors. In 31 st Annual International Symposium on Computer Architecture, June 2004.
- [6] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In International Symposium on Microarchitecture, pages 248–, 1999.
- [7] M. Andersson, J. Cao, M. Kihl, and C. Nyberg. Performance modeling of an apache web server with bursty arrival traffic. In Proceedings of the International Conference on Internet Computing, 2003.
- [8] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In International Symposium on Low Power Electronic Design, August 1998.
- [9] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In Proceedings of 37th Annual Int'l Symposium on Microarchitecture, 2004.
- [10] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. IBM Journal of Research and Development, 44(6):885–898, November 2000.
- [11] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. The International Journal of High Performance Computing Applications, 14(3):189–204, Fall 2000.
- [12] F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in smt processors. Computing Frontiers, pages 433–443, April 2004.

- [13] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05), pages 340–351. IEEE Computer Society, 2005.
- [14] C. Chen, S. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In 10th International Conference on High-Performance Computer Architecture, Madrid, Spain, February 2004. IEEE Computer Society.
- [15] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. Application-specific memory management for embedded systems using software-controlled caches. In Design Automation Conference, pages 416–419, 2000.
- [16] Zeshan Chisti and TJ Vijaykumar. Optimization replication, communication, and capacity allocation in cmps. In Proceedings of the 32rd annual International Symposium on Computer Architecture, 2005.
- [17] SPEC Consortium. Spec website, 2006.
- [18] Juan del Cuvillo, Weirong Zhu, and Guang Gao. Landing openmp on cyclops-64: an efficient mapping of openmp to a many-core system-on-a-chip. In CF '06: Proceedings of the 3rd conference on Computing frontiers, pages 41–50, New York, NY, USA, 2006. ACM Press.
- [19] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. IEEE Micro, 17(5):12–??, /1997.
- [20] A. El-Moursy and D. Albonesi. Front-end policies for improved issue efficiency in smt processors. In Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), page 31. IEEE Computer Society, 2003.
- [21] Matthew K. Farrens and Andrew R. Pleszkun. Strategies for achieving improved processor throughput. In Proceedings of the 18th annual international symposium on Computer architecture, pages 362–369. ACM Press, 1991.
- [22] The Center for Ultra-scale Computing and Information Security(CUCIS) at Northwestern University. Nu-minebench version 2.0, 2006. <http://cucis.ece.northwestern.edu>.
- [23] M. Garcia, J. Gonzalez, and A. Gonzalez. Data caches for multithreaded processors, January 2000.
- [24] Suh G.E.1, Rudolph L., and Devadas S. Dynamic partitioning of shared cache memory. The Journal of Supercomputing, 28(1):7–26, April 2004.
- [25] A. Gonz'alez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In Proc. International Conference on Supercomputing, pp. 338–347, July 1995, 1995.
- [26] Dirk Grunwald and Soraya Ghiasi. Microarchitectural denial of service: insuring microarchitectural fairness. In MICRO, pages 409–418, 2002.
- [27] R. H. Halstead, Jr. and T. Fujita. Masa: a multithreaded processor architecture for parallel symbolic computing. In Proceedings of the 15th Annual International Symposium on Computer architecture, pages 443–451. IEEE Computer Society Press, 1988.

- [28] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufman, San Francisco, California, 1996.
- [29] S. Hily and A. Seznec. Standard memory hierarchy does not fit simultaneous multithreading. In Proc. of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4), 1998.
- [30] Sébastien Hily and André Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. Technical Report PI-1086, IRISA, 1997.
- [31] M. Horowitz, R. Ho, and K. Mai. The future of wires. In Proceedings of the Semiconductor Research Corporation Workshop on Interconnects for Systems on a Chip, May 1999 1999.
- [32] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar K. Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In PACT, pages 13–22, 2006.
- [33] J. Huh, S. W. Keckler, and D. Burger. Exploring the design space of future cmps. In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, October 2001.
- [34] Intel Corporation. IA-32 Intel Architecture Software Developer’s Manual, Volume 1: Basic Architecture. Santa Clara, CA, 2004.
- [35] Intel Corporation. Dual Core Intel Xeon Processor LV Datasheet. Santa Clara, CA, March 2006.
- [36] Intel Corporation. Dual-Core Update to the Intel Itanium 2 Processor Reference Manual For Software Development and Optimization. Santa Clara, CA, January 2006.
- [37] Intel Corporation. Intel Core Duo Processor and Intel Core Solo Processor on 65 nm Process Datasheet, September 2006.
- [38] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In Proceedings of the 24th International Symposium on Computer Architecture, June 1997.
- [39] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In Proceedings of the 17th International Symposium on Computer Architecture, pages 364–373, May 1990.
- [40] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive non-uniform cache structure for wire-delay dominated on-chip caches. In Proceedings of the 10th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 211–222, October 2002.
- [41] D. Kim, S. Liao, P. H. Wang, J. del Cuavillo nad X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel’s hyper-threaded processors. In 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004). IEEE Computer Society, March 2004.
- [42] Seongbeom Kim, Dhruva Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In IEEE PACT, pages 111–122, 2004.
- [43] Rakesh Kumar and Dean M. Tullsen. Compiling for instruction cache performance on a multi-threaded architecture. In Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, pages 419–429. IEEE Computer Society Press, 2002.

- [44] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. IEEE Transactions on Parallel and Distributed Systems, 4(1):41–61, January 1993.
- [45] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. ACM Computing Surveys, 25(3):303–338, 1993.
- [46] Beng-Hong Lim and Ricardo Bianchini. Limits on the performance benefits of multithreading and prefetching. In Measurement and Modeling of Computer Systems, pages 37–46, 1996.
- [47] Chun Liu, Anand Sivasubramaniam, and Mahmut T. Kandemir. Organizing the last line of defense before hitting the memory wall for cmp. In HPCA, pages 176–185, 2004.
- [48] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In ISCA, pages 39–50, 1998.
- [49] Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In Proceedings of the 30th International Symposium on Microarchitecture, pages 114–124, December 1997.
- [50] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. ACM Transactions on Computer Systems, 15(3):322–354, 1997.
- [51] M. Loikkanen and N. Bagherzadeh. A fine-grain multithreading superscalar architecture. In Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques, October 1996.
- [52] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. Ilp versus tlp on smt. Supercomputing, November 1999.
- [53] H.L. Muller, P.W.A. Stallard, and D.H.D. Warren. Hiding miss latencies with multithreading on the data diffusion machine. In Proceedings of the 1995 International Conference on Parallel Processing, pages 178–185, August 1995.
- [54] Hitoshi Oi and N. Ranganathan. Utilization of cache area in on-chip multiprocessor. In ISHPC, pages 373–380, 1999.
- [55] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [56] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for smt processors. Technical report, Department of Computer Science & Engineering University of Washington, Seattle, Washington, 2000.
- [57] M. K. Qureshi, Thompson D., and Patt Y. N. The v-way cache: Demand-based associativity via global replacement. In Proceedings 38rd Annual IEEE/ACM International Symposium on Microarchitecture, December 2005.
- [58] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. PIN: A binary instrumentation tool for computer architecture research and education. In Proceedings of the 2004 Workshop on Computer Architecture Education (WCAE), pages 112–119, June 2004.

- [59] G. Reinman and N. P. Jouppi. Cacti 2.0: An integrated cache timing and power model. Technical report, Hewlett-Packard Labs, Western Research Laboratory, 2000.
- [60] H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, M. van Waveren, and B. Whitney. Large system performance of spec omp2001 benchmarks. In Proceedings of WOMPEI2002, the Workshop on OpenMP: Experiences and Implementations, May 2002.
- [61] Alex Settle and Dan Connors. A dynamically reconfigurable cache for multithreaded architectures. Journal of Embedded Computing Special Edition CMP Architectures, 33(2), December 2005.
- [62] Alex Settle, Josh Kihm, Andy Janiszewski, and Daniel A. Connors. Architectural support for enhanced smt job scheduling. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, 2004.
- [63] Timothy Sherwood, Brad Calder, and Joel S. Emer. Reducing cache misses using hardware and software page placement. In International Conference on Supercomputing, pages 155–164, 1999.
- [64] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In Proceedings of the 17th annual international conference on Supercomputing. ACM Press, 2003.
- [65] B. Sinharoy, R. N. Kalla, J. M. Tandler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. IBM Journal of Research and Development, 49:505–521, July 2005.
- [66] B. Smith. The Architecture of HEP. MIT Press, 1985.
- [67] Allan Snaveley and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In Architectural Support for Programming Languages and Operating Systems, pages 234–244, 2000.
- [68] Evan Speight, Hazim Shafi, Lixin Zhang, and Ram Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture, pages 346–356, Washington, DC, USA, 2005. IEEE Computer Society.
- [69] R. Sree, A. Settle, I. Bratt, and D. Connors. Compiler directed resource management for active code regions. In Proceedings of the Seventh Annual Workshop on Interaction between Compilers and Computer Architectures, February 2003.
- [70] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. IEEE Trans. Comput., 41(9):1054–1068, 1992.
- [71] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In HPCA, pages 117–, 2002.
- [72] Sun Microsystems, Inc. OpenSPARC T1 Microarchitecture Specification. Sunnyvale, CA, 2006.
- [73] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In 22nd Annual International Symposium on Computer Architecture, June 1995.
- [74] Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In Proceedings of the 23rd annual International Symposium on Computer Architecture, pages 191–202, 1996.
- [75] Rollins Turner and Bill Strecker. Use of the lru stack depth distribution for simulation of paging behavior. Commun. ACM, 20(11):795–798, 1977.

- [76] W. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In Proceedings of the 16th Int'l Symp. on Computer Architecture, pages 273–280, June 1989.
- [77] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. SIGARCH Comput. Archit. News, 33(2):336–345, 2005.