Magnus Jahre

Managing Shared Resources in Chip Multiprocessor Memory Systems

Doctoral thesis
for the degree of philosophiae doctor

Trondheim, October 2010

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU
Norwegian University of Science and Technology

Doctoral thesis
for the degree of philosophiae doctor

Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Computer and Information Science

ISBN 978-82-471-2287-7 (printed version)
ISBN 978-82-471-2289-1 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2010:159

Printed by NTNU-trykk

Typeset with LaTeX 2$_\varepsilon$ in Computer Modern 10pt

Abstract

Chip Multiprocessors (CMPs) have become the architecture of choice for high-performance general-purpose processors. CMPs often share memory system units between processes. This may result in independent processes competing for memory bandwidth. Such competition can cause destructive interference which reduces performance predictability, decreases operating system scheduler efficiency and complicates billing for cloud computing providers.

In this thesis, we reduce the effects of these problems by managing miss bandwidth. We use dynamic interference feedback to choose the number of Miss Information/Status Holding Registers (MSHRs) available in last-level private cache of each processor. Furthermore, we provide two different allocation approaches that use this mechanism to improve system performance. The first approach uses simple measurements to decide miss bandwidth allocations and performance feedback to determine if the allocations are beneficial. The second approach selects its allocations based on a miss bandwidth performance model. This model leverages a novel interference measurement scheme called the Dynamic Interference Estimation Framework (DIEF). DIEF provides accurate estimates of the average memory latency a process would experience with exclusive access to all hardware-managed shared resources.

We also investigate the effects of managing memory bandwidth to increase memory bus utilization. Here, we choose prefetches to efficiently utilize the complex DRAM structure of banks, rows and columns. This policy makes prefetches cheaper than demand accesses and increases the performance of processes with predictable access patterns. In addition, efficient prefetch scheduling reduces the degree to which prefetches interfere with the demand accesses of other processes.

Preface

This doctoral thesis was submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree philosophiae doctor (PhD). The work herein was performed at the Department of Computer and Information Science, NTNU, under the supervision of Professor Lasse Natvig.

Acknowledgements

I would like to thank my supervisor Prof. Lasse Natvig for his support, advise and positive attitude through the long process that eventually became this thesis. I also extend my gratitude to my co-supervisors Prof. Mads Nygård and Dr. Torstein Heggebø for providing valuable feedback and encouragement in my evaluation meetings.

Dr. Marius Grannæs and I cooperated on most of the papers in this thesis. I am very grateful for his honest, direct and thought-provoking opinions which significantly increased the quality of the research. I am also grateful to Dr. Haakon Dybdahl for showing me that it is possible to compete at the highest international level. Furthermore, I would like to thank all my co-workers at the Section for Complex Systems for providing a great working environment. I also thank Prof. Otto Anshus at the University of Tromsø for letting me visit his group in the final stages of the work on this thesis.

Finally, I would like to thank my family for their support and patience and Anne for giving me a life outside research.

Magnus Jahre
June 23, 2010

Contents

List of Figures

List of Tables

Abbreviations

ACK	Acknowledgement
AI	Aggregate IPC
AMHA	Adaptive Miss Handling Architecture
ANTT	Average Normalized Turnaround Time
AoC	Age of Computers
ATD	Auxiliary Tag Directory
AWS	Aggregate Weighted Speedup
CMP	Chip Multiprocessor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
C/DC	CZone/Delta Correlation
DC	Delta Correlation
DCPT	Delta Correlating Prediction Tables
DIEF	Dynamic Interference Estimation Framework
DMHA	Dynamic Miss Handling Architecture
DPC	Data Prefetching Championship
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
FAMHA	Fair Adaptive Miss Handling Architecture
FCFS	First Come First Served
FR-FCFS	First Ready - First Come First Served
GHB	Global History Buffer

GSF	Globally Synchronized Frames
HMoS	Harmonic Mean of Speedups
ILP	Instruction Level Parallelism
IP	Interference Point
IPC	Instructions Per Cycle
ITRS	International Technology Roadmap for Semiconductors
JILP	Journal of Instruction Level Parallelism
LLC	Last-Level Cache
LRU	Least Recently Used
MHA	Miss Handling Architecture
MHABC	Miss Handling Architecture Bandwidth Control
MLP	Memory Level Parallelism
MPB	Multiprogrammed Baseline
MSHR	Miss Information/Status Holding Register
MTP	Multiple Time-Sharing Partitions
NACK	Negative Acknowledgement
NFQ	Network Fair Queuing
NoC	Network on Chip
NOTUR	Norwegian Metacenter for Computational Science
NUCA	Non-Uniform Cache Access
OS	Operating System
PC	Program Counter
PC/DC	Program Counter/Delta Correlation
PVC	Preemptive Virtual Clock
PVT	Page Vector Table
RPT	Reference Prediction Table
SMARTS	Sampling Microarchitecture Simulation
SPB	Single Program Baseline
SRP	Scheduled Region Prefetching
STC	Stall Time Criticality

STP	System Throughput
TLP	Thread Level Parallelism
QoS	Quality of Service
VM	Virtual Machine
VPM	Virtual Private Machine
VTMS	Virtual Time Memory System
WAM	Weighted Arithmetic Mean
WHM	Weighted Harmonic Mean

Chapter 1

Introduction

1.1 Chip Multiprocessors (CMPs)

In recent years, general-purpose processor manufacturers have started to provide chips with multiple processor cores [16, 39, 55, 74, 80, 123]. This type of processor is commonly referred to as a multi-core architecture or a *Chip Multiprocessor (CMP)* [113]. CMPs have become a necessity due to four technological and economic trends.

Firstly, high-performance single-core processors consume a great deal of power, and high power consumption necessitates expensive packaging and powerful cooling solutions. This trend effectively limits the maximum power consumption of a processor and is known as the *power wall*. In a CMP, multiple cores can cooperate to achieve high performance at a lower clock frequency and with less aggressive *Instruction Level Parallelism (ILP)* techniques. Designing for a lower clock frequency makes it possible to use a lower supply voltage [112]. Since dynamic power is proportional to the square of the supply voltage, the power reduction can be significant.

Secondly, it has become increasingly difficult to improve performance with techniques that exploit ILP beyond what is common today. Although there is considerable ILP available in the instruction stream [150], extracting it has proven difficult with current process technologies [1]. Thirdly, a processor core is designed once and reused as many times as there are cores on the chip in a CMP. These cores can also be simpler than their single-core counterparts. Consequently, CMPs facilitate design reuse and reduce processor core complexity.

Finally, processor performance has been improving at a faster rate than the main memory access time for more than 20 years [50]. Consequently, the gap between processor performance and main memory latency is large and growing. This trend is referred to as the *memory gap* or *memory wall*. CMPs with multi-threading

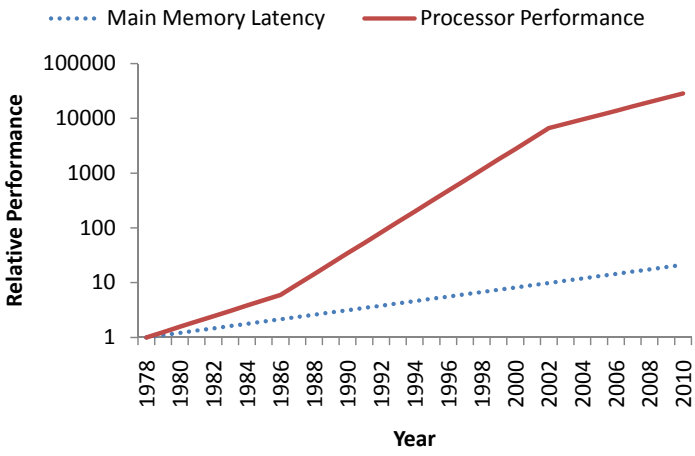


Figure 1.1: The Processor Memory Performance Gap [50]

reduce the impact of this problem by exploiting *Thread Level Parallelism (TLP)* and shifting the focus from single-thread performance to throughput. To achieve the benefits of TLP for a single application, the program must be parallelized. Consequently, programmer effort is required to achieve the performance potential of CMPs for these programs. Furthermore, CMPs often run at a lower clock frequency than their single-core counterparts which can create a one-time reduction of the memory gap.

Figure 1.1 plots the relative improvements in processor performance and main memory latency from 1978 until today. Although CMPs can reduce the impact of the memory wall with a reduced clock frequency, the performance difference is high. As a result, CMPs invest significant resources to hide memory latencies with techniques such as pipelining [13], out-of-order execution [4] and multi-level caches [77, 131, 151]. These latency hiding techniques tend to increase the bandwidth demand [14]. Furthermore, the CMP memory system must provide enough bandwidth to support the needs of an increasing number of concurrent threads. Therefore, CMP memory systems are built to support a significant number of concurrent memory requests [70].

Given these trends, it is likely that future CMPs will need a high-bandwidth memory system. Currently, the limiting factor is the amount of off-chip bandwidth [58]. The amount of off-chip bandwidth available is a combination of the number of I/O pins on the chip and their clock frequency. Figure 1.2 shows the expected improvement in processor performance and off-chip bandwidth as projected by the *International Technology Roadmap for Semiconductors (ITRS)* [62]. To show the trends clearly, both bandwidth and performance are assumed to be 1 in 2007. ITRS expects an average annual improvement of 10% for the number of I/O pins and 25%

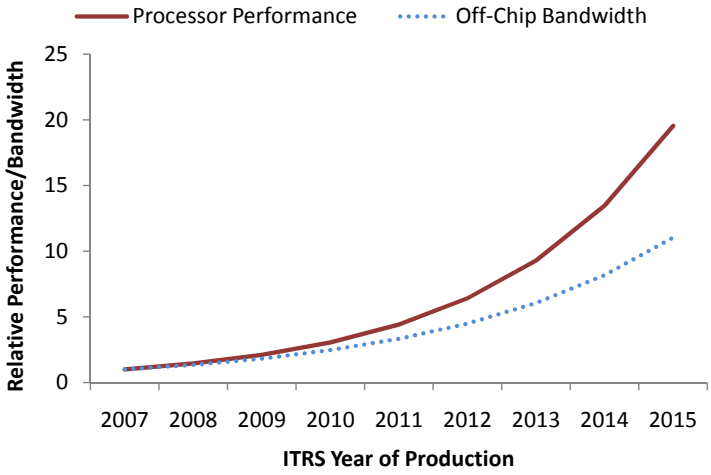


Figure 1.2: Performance and Off-chip Bandwidth [62]

for the off-chip clock frequency. The performance trend combines the expected 5% annual clock frequency improvement with the 40% annual increase in the number of processing cores. Figure 1.2 shows a widening gap between performance and off-chip bandwidth. This gap is not as dramatic as the gap between performance and main memory latency. However, it indicates the need for architectural techniques that improve bandwidth utilization and that such techniques will become increasingly important in the future.

1.2 CMP Shared Memory System Resources

CMPs often share on-chip resources to achieve good resource utilization [17] and fast inter-processor communication [112]. Shared resources are often better utilized than private because they allow resource intensive processes to use all of the available resources. Commonly, CMPs share the on-chip interconnect, *Last-Level Cache (LLC)* and off-chip memory bus. However, the presence of shared resources creates the possibility for *destructive interference* which can reduce overall system performance. Furthermore, destructive interference creates at least four additional problems [102]:

- *Performance predictability* is reduced because the performance of a process depends on the characteristics of the processes it is co-scheduled with. This makes it difficult to analyze software performance.
- System software (*Operating System (OS)* or virtual machine) assumes that the amount of work carried out by a process in one time slice is the same

regardless of which processes are co-scheduled. When destructive interference invalidates this assumption, the efficiency of the OS scheduler is reduced [36].

- Cloud computing systems are emerging as an important arena for CMPs [5]. Here, thousands of distinct users pay for resources and run their applications on a shared computing infrastructure. In this setting, destructive interference causes the undesirable effect that the amount of computation the user gets from a certain investment depends on the activities of other users.

- Finally, malicious programs can deny service to other processes by exploiting the unfairness inherent in shared resource allocation policies [101].

Resource management techniques can provide good utilization of shared resources without the negative effects of destructive interference [106]. These techniques control the division of bandwidth and capacity between running processes. Commonly, this is achieved by making each shared unit thread-aware. In this thesis, we look at the resource allocation problem from a different angle. We improve resource sharing with global bandwidth allocations and leave the allocation policies in each shared unit unchanged.

1.3 Research Questions

There are two important trends facing future CMP architectures. Firstly, the lack of control over resource allocation raises the possibility of destructive interference. The latency impact of interference gets worse as more cores are added to the chip [69]. Secondly, off-chip bandwidth is a limited resource that CMPs should strive to utilize efficiently. The aim of this thesis is to provide techniques that reduce the performance impact of destructive interference and improve the utilization of off-chip bandwidth.

Consequently, the main research question of this thesis is:

How, and at what cost, can performance be improved by managing resource sharing in CMP memory systems?

We approach this question with the following three subquestions:

RQ1 How, and at what cost, can CMP performance be improved by managing *miss bandwidth*?

RQ2 How, and at what cost, can *off-chip bandwidth* management be used to increase CMP performance?

RQ3 How, and at what cost, can the performance and latency effects of resource sharing be *estimated*?

1.4 Thesis Outline

The remainder of this thesis is organized in the following way. First, Chapter 2 presents the theory and techniques related to the contributions in this work before Chapter 3 discusses the methodology used to provide the research results. Then, Chapter 4 presents the research process. In this chapter, research projects that were not included in the final thesis are also discussed. Chapter 5 summarizes the research results and reviews the papers given the benefits of hindsight. Finally, Chapter 6 summarizes the contributions, discusses how the contributions are related to the research questions and provides some indications of further work. In addition, a short discussion of how the challenges confronted in this thesis may change in the future is included.

Chapter 2

Background

This chapter provides the background necessary to understand the papers included in this thesis. To achieve this, I introduce the main research results that our contributions build on. In addition, I discuss other approaches to the problems targeted by this thesis. Section 2.1 provides the necessary background on performance measurements. This information is needed to understand the discussion of CMP shared resources (Section 2.2) and full-system resource management techniques (Section 2.3). In this thesis, we also use prefetching to better utilize off-chip bandwidth. Therefore, Section 2.4 introduces the hardware prefetching heuristics used in this work as well as discussing memory controller prefetch scheduling and prefetching in CMPs.

2.1 Quantifying CMP Performance

2.1.1 Measuring Performance

Hennessy and Patterson [50] argue that the only consistent and reliable measure of computer performance is execution time. They define *Wall Clock Time* as the latency to complete a program including operating system tasks, disk accesses and other input/output activities. Furthermore, the wall clock time of a process may depend on the activities of other processes in a multiprogrammed system. This may not be appropriate in all contexts. Therefore, they define *CPU Time* to only include the time the processor uses to execute the instructions of the process.

$$\text{CPU Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Clock Cycles}}{\text{Instructions}} \cdot \frac{\text{Seconds}}{\text{Clock Cycles}} \tag{2.1}$$

Equation 2.1 breaks down CPU Time into three components. In simulator-based computer architecture studies, it is common to run the same sequence of instructions from a program on different architectural configurations. Furthermore, it is common to keep the number of seconds per clock cycle (i.e. the clock cycle time) constant. Under these assumptions, the architectural changes will only affect the number of *Cycles Per Instruction (CPI)*. Therefore, Equation 2.1 can be simplified for such studies:

$$\text{CPU Time} \propto \frac{\text{Clock Cycles}}{\text{Instructions}} \tag{2.2}$$

Computer architecture research often focuses on improving performance. Since performance is the inverse of execution time, it is common to report the inverse of CPI. This quantity is commonly referred to as *Instructions Per Cycle (IPC)*. CPI and IPC measurements can be misleading if the program commits instructions without making forward progress (e.g. in busy-wait loops) [2].

High performance has a slightly different meaning for a user and a system administrator. The user is interested in a low *turnaround time* (i.e. the time from starting a task until its completion) and a low *response time* (i.e. the time from starting a task until it produces its first output). For the system administrator, it is more important to maximize the total amount of work than to minimize the latency of any individual task. The amount of work carried out per unit time is called *throughput*. Unfortunately, it may not be possible to simultaneously provide both high throughput and low turnaround time. Consequently, architectural techniques often need to achieve a good trade-off between these notions of performance.

$$\text{Speedup} = \frac{P_{\text{technique}}}{P_{\text{baseline}}} \tag{2.3}$$

When comparing architectural techniques, the performance of the new technique relative to some baseline is often more interesting than its absolute performance. This quantity is called *speedup* and illustrated by Equation 2.3. Here, $P_{\text{technique}}$ is the performance with a particular architectural enhancement and P_{baseline} is often the performance without it. For instance, $P_{\text{technique}}$ may be performance with a particular prefetcher and P_{baseline} performance without prefetching.

2.1.2 Aggregating Performance

General-purpose processors aim to achieve good performance for a variety of programs. Consequently, it is common to evaluate the performance of architectural techniques for a number of different programs. These programs are called *benchmarks*, and a collection of benchmarks is referred to as a *benchmark suite*. This approach has created the need for aggregating performance across a benchmark

Table 2.1: Aggregate Performance Alternatives

Name	Formula
Weighted Arithmetic Mean	$\text{WAM} = \frac{1}{n} \sum_{i=1}^{n} \omega_i \cdot P_i$
Weighted Harmonic Mean	$\text{WHM} = \frac{n}{\sum_{i=1}^{n} \frac{\omega_i}{P_i}}$
Geometric Mean	$G = \sqrt[n]{\prod_{i=1}^{n} P_i}$

suite. The most common alternatives for performance aggregation are shown in Table 2.1.

Correctly aggregating performance with a single number has proven to be difficult [50, 72, 73, 132]. John [72] proved mathematically that the *Weighted Arithmetic Mean (WAM)* and *Weighted Harmonic Mean (WHM)* can be used interchangeably as long as the appropriate weights are chosen. For metrics that are ratios (e.g. $\frac{A}{B}$), the unweighted arithmetic mean is valid if B is equally weighted. Conversely, the unweighted harmonic mean is valid if A is equally weighted. With IPC, this implies the arithmetic mean is valid if the number of clock cycles are kept constant.

Smith [132] advises that the geometric mean should be avoided when aggregating performance measurements. In general, the geometric mean is most convenient when component quantities are multiplied while the harmonic and arithmetic means are convenient when they are added [73]. However, the geometric mean has the useful property that the baseline cancels out from the ratio of two geometric means of speedups relative to a common baseline [50]. In this thesis, the geometric mean is not used.

2.1.3 Quantifying the Performance Impact of Interference

To establish the performance effects of destructive interference it is helpful to compare to a configuration where interference cannot occur. Figure 2.1 shows two possible ways to create such baselines. The first, called a *Multiprogrammed Baseline (MPB)*, is created by statically dividing each shared resource equally between threads [18, 56, 70]. With MPB, the baseline results can be collected for all benchmarks in a workload with a single simulator run. Conversely, the process is run alone with exclusive access to all shared resources in a *Single Program Baseline (SPB)* [70, 93, 103, 153]. In this case, one simulator run is necessary for each benchmark to collect the baseline results. The baseline configuration (i.e. either SPB or MPB) is referred to as the *private mode*. Conversely, the processes compete for resources in the *shared mode*.

The performance effects of CMP resource sharing are commonly described with two

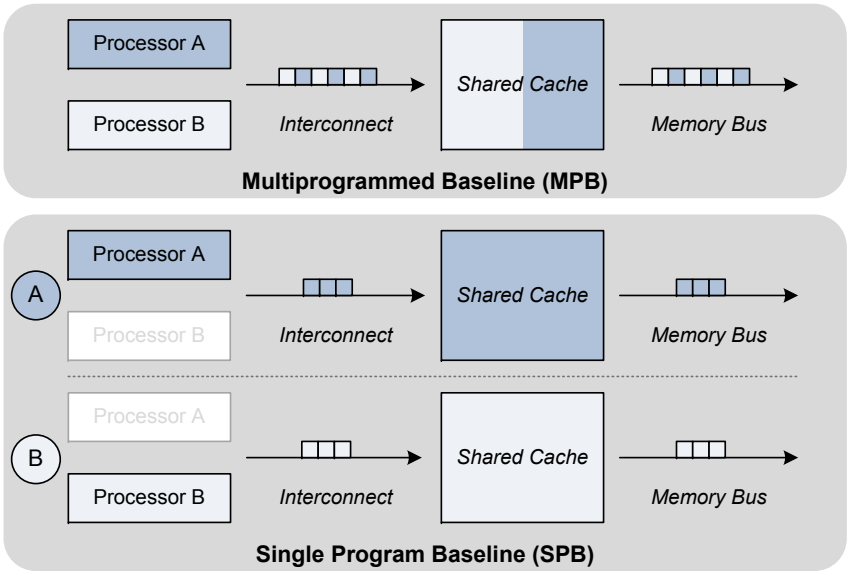


Figure 2.1: Resource Allocation Baselines (Private mode)

concepts. Firstly, the performance reduction due to memory system interference should affect all processes equally [79]. This concept is referred to as *fairness*. Secondly, a resource allocation implementation can strive towards putting a limit on the maximum performance reduction of a process. This is known as *Quality of Service (QoS)* [18]. It is not always possible to simultaneously achieve good fairness/QoS and throughput [56]. Consequently, a resource allocation technique should strive to provide a balance between these notions of performance [34].

2.1.4 System Performance Metrics

$$\text{Shared Mode Speedup} = \frac{P}{\mathcal{P}} \tag{2.4}$$

Equation 2.4 shows the speedup of the shared mode performance P relative to the private mode performance $\mathcal{P}$. The shared mode speedup is the central component of many system performance metrics. In general, shared mode quantities are denoted with regular letters (e.g. P) and private mode quantities with calligraphic letters (e.g. $\mathcal{P}$). With SPB, the shared mode speedup is a number between 0 and 1.

Table 2.2 shows four common CMP system performance metrics. Eyerman and Eeckhout [34] showed that *Harmonic Mean of Speedups (HMoS)* [93] is the inverse of *Average Normalized Turnaround Time (ANTT)* while *Aggregate Weighted Speedup (AWS)* [133] represents system throughput. In other words, HMoS is a

Table 2.2: Multiprogrammed Workload Performance Metrics

Metric	Formula	System-Level Meaning	
AWS or STP	$\sum_{p=0}^{n} P_p/\mathcal{P}_p$	System Throughput	[133]
HMoS	$\frac{n}{\sum_{p=0}^{n} P_p/\mathcal{P}_p}$	Inverse of *Average Normalized Turnaround Time (ANTT)*	[93]
Fairness	$\frac{min(P_i/\mathcal{P}_i)}{max(P_j/\mathcal{P}_j)}$ $i,j \in \{0,n\}$	Assumed by system software	[38]
AI	$\sum_{p=0}^{n} P_p$	None [34]	-

user oriented metric and AWS is a system oriented metric. The AWS metric is also referred to as the *System Throughput (STP)* metric while HMoS can be called *fair speedup*.

System software assumes that the forward progress of a process is the same regardless of which processes it is co-scheduled with. If we assume equal priorities, this assumption is true if the performance reduction due to sharing effects is distributed equally among threads. This is measured by the *fairness metric* [38]. With SPB, the fairness metric is a number between 0 and 1 where 0 means that at least one process is not making forward progress while 1 means that the slowdown is perfectly distributed.

Aggregate IPC (AI) measures raw IPC throughput and is maximized by prioritizing the processes that can achieve high IPC numbers. These are likely the processes that are least impacted by memory system interference. For this reason, AI should not be used to measure throughput since all architectural techniques should ensure the forward progress of all processes [93, 133].

2.2 CMP Shared Resources

Figure 2.2 shows a possible CMP memory system where each core has a small private Level 1 (L1) cache. Since accessing the L1 cache is often on the critical path of the processor, the hit latency is limited to a few processor cycles. This limits the size of the L1 cache, and a L2 cache is added to increase the amount of data stored close to the processor core. Then, the private memory systems are connected to a large shared L3 cache with an on-chip interconnect. Finally, one or more memory controllers manage the high-speed interface to main memory. This example architecture is similar to the recent i7 processor from Intel [16].

In Figure 2.2, the processor cores share the on-chip interconnect, the shared cache and the off-chip memory bus, and inter-process interference can occur in all of these units [69]. Inter-process interference is due to interleaving of requests from different processes which may increase queuing latencies and reduce shared cache and memory bus locality. This results in higher memory latencies, but the perfor-

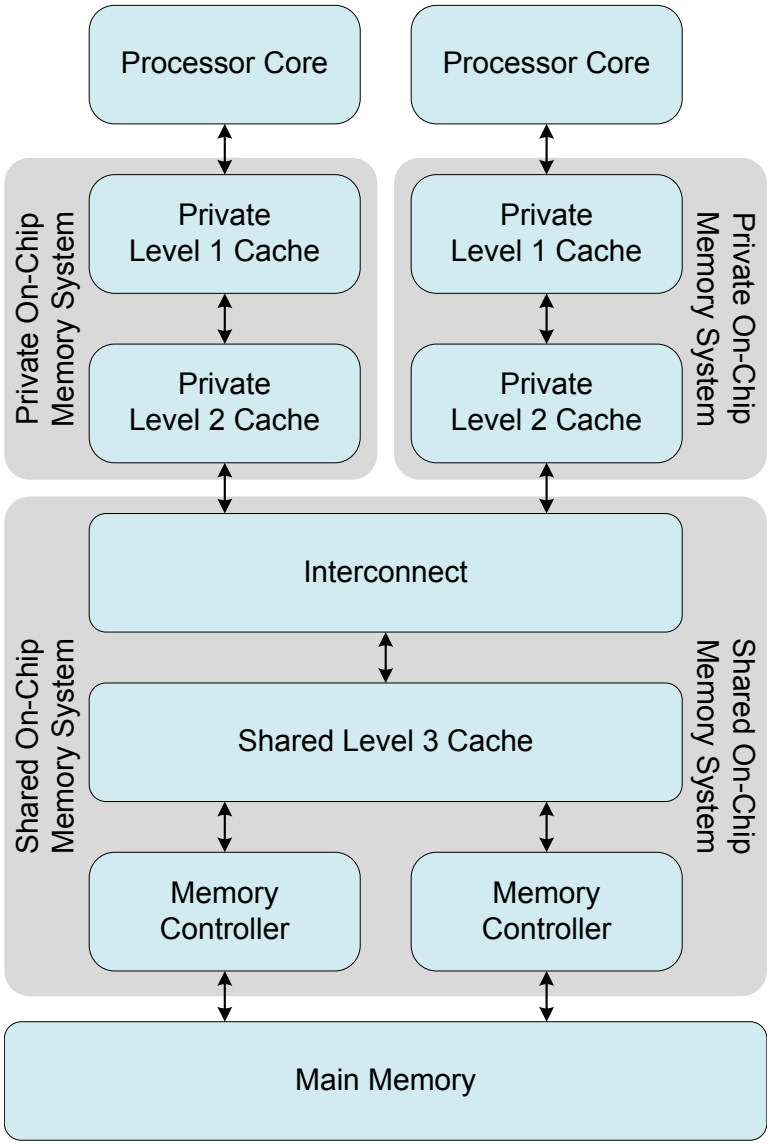


Figure 2.2: Chip Multiprocessor Memory System Example

mance impact of interference depends on the ability of the process to utilize the latency hiding mechanisms of the processor core and memory system. Resource management implementations aim to reduce this performance impact. Formally, interference is defined as [70, 102]:

$$I = L - \mathcal{L} \tag{2.5}$$

In this equation, interference I is the difference between the shared mode memory latency L and the private mode memory latency $\mathcal{L}$. In other words, interference is the additional latency a process experiences due to memory system sharing effects.

Nesbit et al. [106] divide CMP resource management systems into three subsystems:

- *Feedback Mechanisms* – These mechanisms measure the sharing or performance related metrics that the policies can use to make partitioning decisions.
- *Allocation Policies* – Based on the metrics provided by the feedback mechanisms, the allocation policy chooses a resource partitioning that maximizes a certain performance metric.
- *Partitioning/Allocation Mechanisms* – The partitioning mechanisms enforce the resource allocations selected by the policy.

The mechanisms provide the primitives that the policies can be built upon. These mechanisms may need to be implemented in hardware if they are sufficiently tightly coupled to the hardware units they measure or control. For the allocation policies, flexibility is important because CMPs are used in a variety of contexts. This can be achieved by implementing the allocation policies in software [106]. However, this limits the frequency with which resources may be reallocated so a combined hardware/software approach may be necessary.

It is also possible to reduce the performance effects of interference by making the OS scheduler interference aware [36, 37]. Here, processes that suffer from interference are allowed to use the CPU for longer than other processes. This approach has the advantage that it can be implemented without hardware support. Since our primary focus is on hardware support for resource management, scheduling techniques are not investigated further.

2.2.1 Shared Cache

CMPs have large on-chip caches to reduce the performance impact of the performance gap between the processor and main memory. Caches rely on the fact that most programs exhibit *spatial* and/or *temporal locality*. Temporal locality is the tendency that a recently used data element will be used again in the near future, while spatial locality is the tendency for accessing data elements with memory addresses close to the recently used elements.

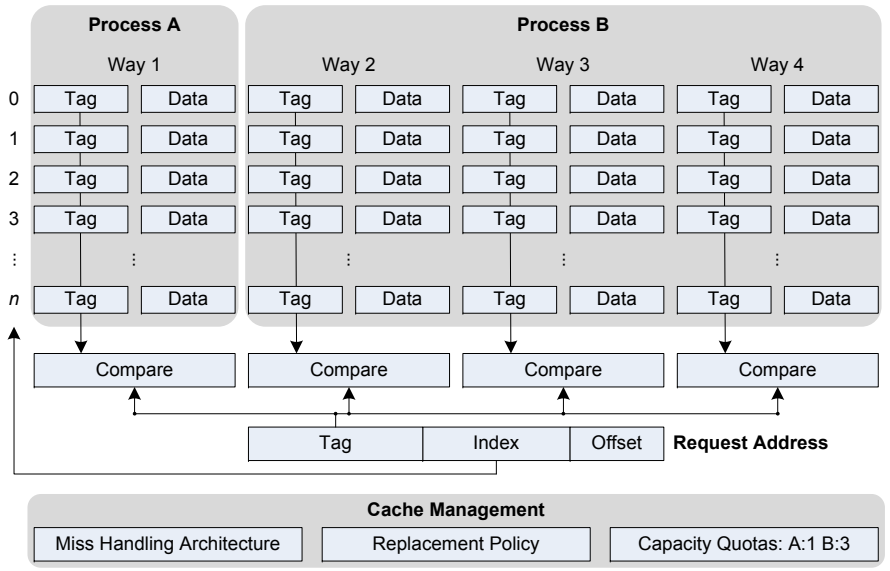


Figure 2.3: 4-way Set-Associative Shared Cache Example

Figure 2.3 illustrates the high-level structure of a shared 4-way set associative cache. In this cache, each data element, called a cache *block* or *line*, can be stored in four possible locations. These four locations are referred to as a *set*. The set of a memory request is determined by the *index* part of the request address. When searching for a cache block, the *tags* of all blocks in the set are examined. The tag is the part of the address that is not part of the index or the cache block offset. This search can be carried out in parallel for low access latency or serially to conserve energy [143].

Traditionally, cache misses have been classified according to the 3C model [53, 54]. In this model, a cache miss is either a compulsory, conflict or capacity miss. A compulsory (or cold start) miss is the first access to a block, and for these misses the size of the cache does not matter. A conflict miss is a miss that occurs because of limited set associativity. In other words, these misses would not occur if every address could be stored in every cache block (i.e. in a fully associative cache). Finally, a capacity miss occurs because the cache is too small to store the data needed by the running process.

2.2.1.1 Miss Handling Architectures

The *Miss Handling Architecture (MHA)* is the cache subsystem that manages cache misses in a *non-blocking* or *lockup-free* cache. A non-blocking cache [81] can continue to service requests while misses are being serviced by units further down in

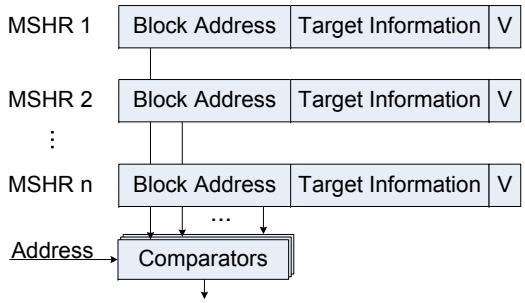


Figure 2.4: Generic *Miss Information/Status Holding Register (MSHR)* File

the memory hierarchy. In addition, a non-blocking cache combines accesses to the same cache block such that only one request is sent. Multiple misses to a single cache line can happen frequently in out-of-order processors since a cache block is often significantly larger than a processor word. The first miss to a cache block is called a *primary miss* and subsequent misses to cache blocks are referred to as *secondary misses* [35].

The main hardware structure in an MHA is the *Miss Information/Status Holding Register (MSHR)*. Commonly, n MSHRs are combined into an MSHR file which makes it possible for the cache to handle n concurrent misses without blocking. A blocked cache cannot service requests. There may be multiple MSHR files in a cache with multiple banks. Consequently, a *Miss Handling Architecture (MHA)* consists of one or more MSHR files [146].

Figure 2.4 shows a generic MSHR file. Here, each MSHR contains a block address, target information and a valid bit. The block address is the address of the miss and the valid bit signifies that the MSHR is in use and contain data. If all valid bits are set, the cache must block since there are no storage available for additional misses. The target information field stores the miss information that is not needed further down in the memory hierarchy. Furthermore, its organization determines which combinations of secondary misses that can be handled without blocking. Kroft [81] used an *implicit* target organization where target storage is allocated for each processor word. Consequently, the cache can only handle one access to a processor word before it must block. Farkas and Jouppi [35] discussed three other target storage implementations. With *explicit* targets, the processor word offset is stored explicitly which makes it possible to have multiple misses to the same word without blocking. In addition, they discussed the aggressive inverted MHA where miss information storage is allocated for all possible destinations of fetch data. Finally, miss information can be stored in the pending cache line.

Recently, there has been some research targeting MHAs for unconventional architectures. Tuck et al. [146] proposed the Hierarchical MHA that provides enough

miss parallelism to meet the bandwidth demands of high *Memory Level Parallelism (MLP)* processors. In addition, Loh [92] provides a high-capacity MHA for 3D-stacked memory architectures.

2.2.1.2 Shared Cache Management

In a shared cache, concurrently running processes can evict each others cache blocks. An *interference miss* is defined as a cache miss that occurs in the shared mode but not in the private mode [70]. In addition, Srikantaiah et al. [139] recently proposed the CII model which complements the 3C model. In this model, a cache miss is classified as a compulsory, inter-processor or intra-processor miss. The compulsory miss category is the same as in the 3C model. The intra-processor misses are the misses that are due to a cache block being evicted by the same processor that owns the cache block. Conversely, inter-processor misses are due to a different processor evicting the block. An inter-processor miss is not necessarily an interference miss. The reason is that the block could have been evicted by the owner if the other processor had not evicted it first.

Most cache partitioning techniques use some form of way-partitioning where each *way* of a set associative cache is allocated to a process. A way is sometimes referred to as a *column*. In Figure 2.3, process A has been allocated one way and process B has been allocated three ways. Way-partitioning can be implemented in at least two different ways. Firstly, it is possible to enforce that the cache blocks of a process are limited to using a subset of the columns [20, 100, 124]. Secondly, the replacement policy can be modified such that each process only is allowed to occupy a certain number blocks in a set [18, 29, 47, 63, 110, 116, 118]. In this case, the blocks can be physically stored in any of the columns.

Xie and Loh [153] showed that the cache can be managed by controlling the insertion and promotion of cache blocks. Normally, new cache blocks are inserted at the most recently used position which maximizes their lifetime in the cache. Furthermore, a block is normally promoted to the most recently used position when it is accessed. By inserting and promoting blocks to positions closer to the least recently used position, Xie and Loh implicitly partitions the cache among cores.

To establish cache quotas, it is helpful to estimate the cache capacity allocation sensitivity of a process. Since the cache tag array is much smaller than the data array, additional tag arrays can be added at a moderate storage cost. This technique is based on the observation that the cache tags define which blocks are available in the cache. An additional tag array is commonly referred to as an *Auxiliary Tag Directory (ATD)* or a *shadow tag* [30, 117]. *Set sampling* is often employed to reduce the storage overhead of ATDs. Here, the cache is divided into regions called *constituencies* [117]. One set in each constituency is chosen as a *leader set* and the other sets become *follower sets*. By assuming that the leader set events are representative for the follower sets, it is only necessary to store the leader sets in the ATD.

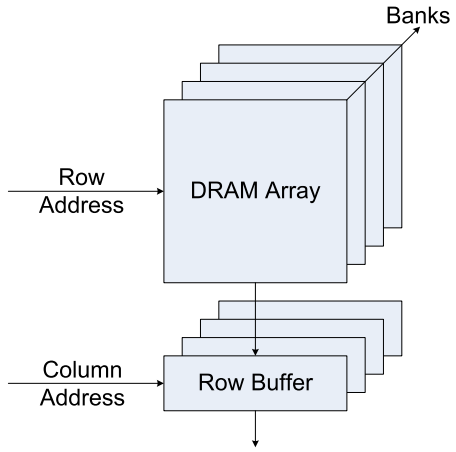


Figure 2.5: The 3D Structure of DRAM

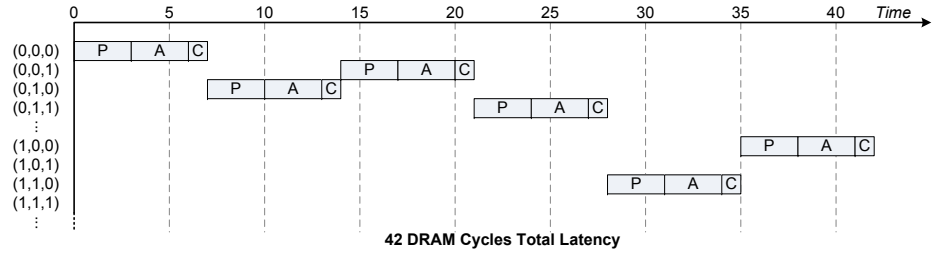
2.2.2 Memory Bus and DRAM

In a traditional random access device, all locations have the same latency. However, modern *Dynamic Random Access Memory (DRAM)* devices do not fully comply to this definition. In order to maximize bandwidth, DRAMs are commonly organized as a 3D matrix of bits with dimensions of *rows*, *columns* and *banks*. Figure 2.5 illustrates this organization.

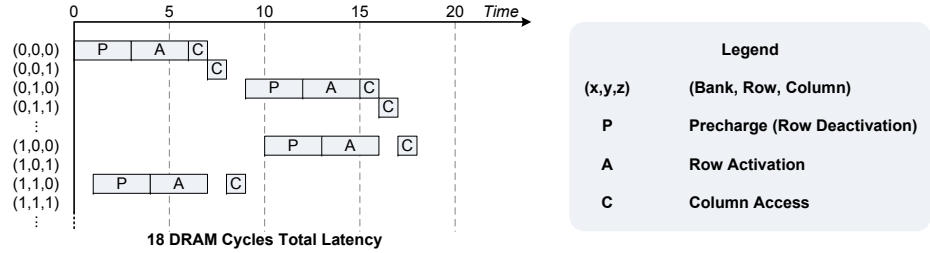
Commonly, a DRAM read transaction consists of first sending the row address, then the column address and finally receiving the data. When a row is accessed, its contents are stored in a register known as the row buffer, and a row is often referred to as a *page*. The row buffer is commonly much larger than a cache line to leverage the internal bandwidth of the DRAM module. If the row has to be activated before it can be read, the access is referred to as a *row miss* or *page miss*. It is possible to carry out repeated column accesses to an open page, called *row hits* or *page hits*. This is a great advantage as the latency of a row hit is much lower than the latency of a row miss [120]. The situation where two consecutive requests access the same bank but different rows is known as a *row conflict* and is very expensive in terms of latency. DRAM accesses are pipelined, so there are no idle cycles on the memory bus if the next column command is sent while the data transfer is in progress. Furthermore, command accesses to one bank can be overlapped with data transfers from a different bank.

2.2.2.1 DRAM Scheduling

The 3D structure of rows, columns and banks makes DRAM subsystem throughput depend heavily on the order of memory requests. Therefore, throughput can be



(a) Accesses Scheduled in Arrival Order



(b) Accesses Reordered with the FR-FCFS Algorithm

Figure 2.6: Simplified DRAM Access Reordering Example [120]

improved considerably by dynamically reordering requests to improve page locality. Rixner et al. [120] provide a convincing example that illustrates the benefits of out-of-order DRAM scheduling (Figure 2.6).

Figure 2.6(a) shows a memory access order that poorly utilizes the parallelism available in the DRAM subsystem. In this example, we assume a 3 DRAM cycle precharge latency, a 3 cycle row activation latency and 1 cycle column access latency. In this example, servicing requests in the arrival order results in a total latency of 42 DRAM cycles. Figure 2.6(b) illustrates the possible latency improvement from reordering. Firstly, we take advantage of that command accesses to different banks can be carried out in parallel. Secondly, we carry out all pending column accesses when the row is active. These two optimizations reduce the total latency to 18 DRAM cycles and improves memory bus utilization from 14% to 33%. Note that only one command can occupy the memory bus at a time and that precharge and activate commands are sent in their first cycle.

Rixner et al. [120] showed that reordering can be implemented with three rules:

1. Prioritize requests that can be issued in this cycle (i.e. *ready* commands) over commands that are not ready
2. Prioritize column commands over other commands
3. Prioritize older commands over younger commands

This scheduling algorithm is commonly referred to as *First Ready - First Come First Served (FR-FCFS)* scheduling, and a few researchers have proposed additions to it. Shao and Davis [125] proposed burst scheduling which clusters accesses to the same row into bursts. Furthermore, they prioritize reads over writes, but writes are piggybacked on read bursts to reduce the probability of blocking due to a full write queue. Zhu and Zhang [158] observed that performance could be improved further by taking *criticality* into account. A criticality-aware scheduler prioritizes the requests that contain the words that the processors are currently waiting for. Finally, Shao and Davis [126] observed that bank conflicts can be minimized by cleverly choosing the address mapping of banks, rows and columns.

2.2.2.2 Process-Aware DRAM Scheduling

The FR-FCFS scheduling algorithm does not differentiate between requests from different processes. Consequently, a process with good page locality can significantly delay the requests of other processes. To avoid this, Nesbit et al. [109] adapted network fair queuing for use in DRAM scheduling. Network fair queuing was originally used to provide fairness in packet switched networks [40]. Nesbit et al. augmented the DRAM scheduler with a model of a *Virtual Time Memory System (VTMS)* (one per process). Each VTMS is allocated a certain fraction of the bandwidth available in the real system. This fraction determines the bandwidth allocation of the process. Then, the finish time of the request in the VTMS is used instead of the shared mode arrival time in rule 3 of the FR-FCFS scheduling algorithm. Furthermore, Nesbit et al. limit the amount of reordering to avoid that a process with good page locality significantly delays requests with lower virtual finish times from other processes.

Network fair queuing distributes DRAM bandwidth in a fair manner. However, fairly distributing DRAM bandwidth does not necessarily result in a fair division of DRAM *latency* [102, 119]. Rafique et al. [119] extended the work of Nesbit et al. [109] by using an adaptive technique to tune bandwidth shares to achieve the desired average latencies. In addition, Mutlu and Moscibroda [102] provide a scheduler that equalizes the memory related stall time of different processes. Finally, Iyer et al. [64] showed that the reordering mechanism can be used to differentiate between priority classes by letting requests from high priority processes bypass the requests of low priority processes.

The schedulers discussed so far augment the FR-FCFS algorithm with additional rules. Mutlu and Moscibroda [103] approach the problem from a different angle with their batch scheduling technique. Here, they create batches of requests based on arrival time and which processor the request originated in. A batch is a group of requests with a limited size, and the batch containing the oldest request is serviced before other batches to avoid starvation and provide fairness. Within a batch, requests are scheduled to preserve bank parallelism which improves throughput. In addition, Ipek et al. [60] showed how machine learning could be applied to the access scheduling problem.

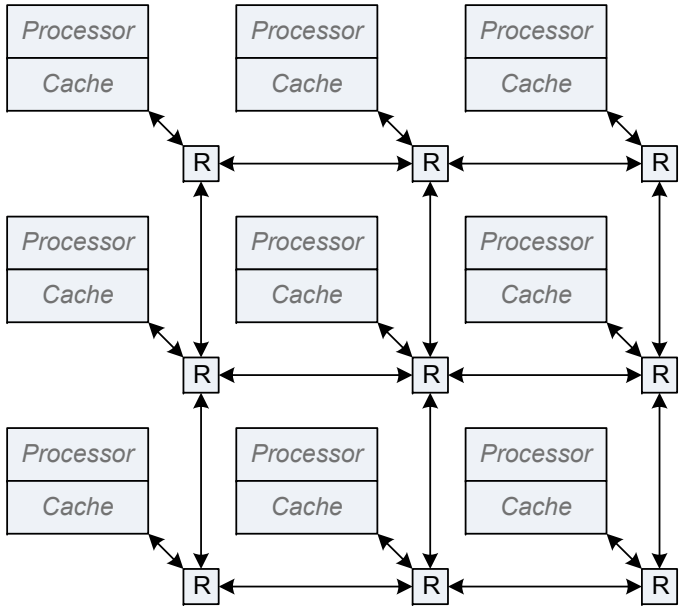


Figure 2.7: 3x3 Mesh Network on Chip

2.2.3 On-Chip Interconnect

The on-chip interconnect can be constructed in two main ways: by ad-hoc global wiring or with an on-chip network or *Network on Chip (NoC)* [24]. For a general-purpose processor, a NoC has several advantages. Firstly, it uses the available wires more efficiently since the wires are not necessarily idle when only a few of the end-points are using the network. Secondly, it adds structure and modularity to the chip which is helpful during the design process.

There has been a large amount of research on interconnects for high performance computers [23]. It is likely that much of this knowledge can be used when designing efficient on-chip networks. However, high-performance computer interconnects have traditionally been limited by the number of available wires and pins [24]. For NoCs, design trade-offs change since wires are abundant and buffer space is limited. Figure 2.7 shows a possible CMP NoC.

This difference has lead researchers to propose QoS techniques that take advantage of the available wires and reduce the need for buffer space. Recently, Lee et al. [87] proposed *Globally Synchronized Frames (GSF)*. In GSF, packet classification is carried out at the source nodes to reduce management costs and avoid significantly increasing router complexity. They achieve QoS by assembling packets into *frames*. Each frame contains a limited number of *flow control units (flits)*, and flits in the current head frame have higher priorities than flits with other frame numbers. The

identity of the current head frame is communicated through a dedicated barrier network. This global communication is an example of a feature that can be added when the network is located on a single chip.

Although GSF only slightly increases router complexity, it does require large source buffers to achieve good throughput [46]. Furthermore, it can have low throughput under traffic patterns which only saturate a part of the network due to its global assignment of frame numbers. Grot et al. [46] recently proposed *Preemptive Virtual Clock (PVC)* which alleviates these problems. PVC tracks bandwidth consumption over a time interval and uses this to prioritize packets. Consequently, it avoids per-flow buffering in the network and large buffers in source nodes. Since there is no per-flow buffering in the routers, *priority inversion* is possible. Priority inversion is the situation where low priority flits impede the progress of flits with higher priorities. To avoid this, PVC detects priority inversion and preempts the low priority flits causing it. This is communicated to the source with a *Negative Acknowledgement (NACK)* which causes the source to resend the data. ACK and NACK messages are sent on a dedicated network.

Das et al. [25] observed that the performance impact of an on-chip bandwidth allocation depends on process characteristics. They coined the term *Stall Time Criticality (STC)* to describe the performance impact of increased memory latencies. The STC of a process depends on three phenomena. Firstly, the cost of a miss can be amortized by serving multiple misses in parallel. This is known as *Memory Level Parallelism (MLP)*, and more MLP tends to reduce STC. Secondly, high average memory latency (e.g. many off-chip accesses) tend to increase STC. Finally, processes differ in their bandwidth demand from the network. Requests from processes with a high demand tend to be less critical than requests from low-demand processes. Das et al. observed that the number of private cache misses per instruction could be used to indicate the STC of a process. Then, requests from high STC processes are given higher priorities than processes with low STC.

2.3 Full-System Resource Management

So far, the focus has been on resource management solutions that target specific shared units in the CMP memory system. However, a system wide approach is needed to provide fairness or QoS for the complete hardware-managed memory system since a good partitioning for one shared unit can have adverse effects in other units [87]. For instance, poor cache partitioning can increase memory bus queuing latencies enough to reduce overall performance [138]. These approaches can be divided into two categories. In the first approach (Section 2.3.1), each shared unit is extended with enforcement and feedback mechanisms. Then, a central policy collectively manages the allocations for all shared units. In the second approach (Section 2.3.2), the shared units are only extended with measurement mechanisms. Then, the memory access frequency of one or more of the processors is modified to improve resource sharing.

Liu et al. [91] provide an analytical model of the interaction between cache capacity and off-chip bandwidth allocations. To derive the model, they combined an additive CPI model with queuing theory. They found that the benefits of off-chip bandwidth partitioning are closely tied to the processes' shared cache miss rate differences. This is based on the observation that an unmanaged memory bus partitions requests according to their access frequencies. In addition, they observed that cache partitioning can impact off-chip bandwidth sharing in two ways. Firstly, cache partitioning can reduce the amount of off-chip traffic and thereby increase the amount of bandwidth available. Cache partitioning can also reduce or increase the need for bandwidth partitioning by changing the relationship between off-chip access frequencies.

2.3.1 Coordinated Resource Allocations

Nesbit et al. [106] proposed to divide the shared memory system by creating a *Virtual Private Machine (VPM)* for each process or *Virtual Machine (VM)*. A VPM is a specification of the micro-architectural resource requirements of a process and is divided into minimum and maximum VPMs. Hardware mechanisms are provided to ensure that a process receives at least the amount of resources specified in its minimum VPM. If there are excess resources, these are distributed to achieve secondary performance objectives or improve resource utilization.

Iyer et al. [64] provides best-effort *Quality of Service (QoS)* by providing resources to one or more high priority processes. To control their technique, they define two metrics: QoS Targets and QoS Constraints. The QoS Target is the performance goal for a high priority application while the QoS Constraint is the lowest acceptable performance for a low priority application. The QoS Targets and QoS Constraints can be provided by online monitors [70, 156] or offline profiling. Iyer et al. [64] control cache space allocations with a modified *Least Recently Used (LRU)* cache replacement policy. Memory bandwidth is managed by letting requests from a high priority process bypass low priority requests. In addition, Bitirgen et al. [12] used artificial neural networks to control memory system resource allocations and power consumption.

2.3.2 Rate-Based Resource Management

Herdrich et al. [52] provides QoS by reducing the memory system access frequency of a low priority process if it interferes with a high priority process. This approach is referred to as *rate-based* resource management. Herdrich et al. investigated two enforcement mechanisms: frequency scaling and clock modulation. They found that frequency scaling was not very effective at providing QoS since memory intensive processes spend most of their time waiting for memory. Consequently, they were able to cause interference both at high and low clock frequencies. With clock

modulation, the clock is gated for short intervals. While the clock is gated, the processor will not carry out any work. This effectively reduces the access frequency for both memory intensive and processor intensive processes. In addition to providing QoS, clock modulation also reduces power consumption.

Ebrahimi et al. [32] recently proposed source throttling which uses two enforcement mechanisms to provide rate-based resource management. The first enforcement mechanism manipulates the number of MSHRs available in each core's last-level private cache. Secondly, they control the frequency with which a core is allowed to inject requests into the shared memory system. The resource allocations are chosen based on runtime interference estimates.

2.4 Hardware Prefetching

Prefetching is the task of fetching data and/or instructions into the on-chip caches before they are requested by the processor. Consequently, prefetching can have a considerable performance impact since the cache access latency is significantly smaller than the DRAM access latency. A prefetcher commonly analyses the access stream to find patterns that can be used to predict future accesses. However, prefetching is a speculative technique and fetching the wrong data can lead to cache pollution and wasted bandwidth.

It is helpful to categorize prefetches as *good* or *bad*. A prefetch is *good* if the prefetched cache block is referenced by the application before it is replaced [142]. If block is replaced without being used, it is classified as *bad*.

$$\text{Accuracy} = \frac{\text{Good Prefetches}}{\text{Good Prefetches} + \text{Bad Prefetches}} \tag{2.6}$$

Equation 2.6 defines the *accuracy* metric which measures how often the prefetcher's prediction is correct [142]. For prefetches to be useful, they also need to be issued ahead of time to ensure that the data is available in the cache when the processor needs it. This property is known as *timeliness*.

$$\text{Coverage} = \frac{\text{Good Prefetches}}{\text{Cache Misses Without Prefetching}} \tag{2.7}$$

Unfortunately, issuing highly accurate and timely prefetches are not necessarily sufficient to provide a significant performance improvement. In addition, a considerable portion of the cache misses of a process needs to be removed. This is measured by the *coverage* metric which is defined in Equation 2.7 [142]. Increasing coverage often reduces accuracy, and a good prefetching scheme should therefore strive to achieve a balance. The main advantage of the accuracy and coverage metrics are that they are simple to use and easy to understand. However, they

PC Address	Last Address	Stride	State

Figure 2.8: Reference Prediction Table Entry Format

do not always accurately explain why prefetching can lead to increased bandwidth usage or cache pollution [142].

There is a significant amount of research that proposes new prefetching heuristics [115]. In this thesis, prefetching is used to improve off-chip bandwidth utilization. Since the choice of prefetching heuristic is largely orthogonal to these techniques, Section 2.4.1 only discusses the heuristics used in the included papers. Then, Section 2.4.2 discusses how the prefetcher can be integrated with the memory controller. Finally, Section 2.4.3 discusses how multiple prefetchers can be coordinated in a CMP. A through review of prefetching is provided by Grannæs [42].

2.4.1 Hardware Prefetch Heuristics

The simplest prefetching heuristic is *sequential* prefetching [130]. In its simplest form, sequential prefetching fetches the next cache block in ascending address order when a cache block is accessed. This policy can be very effective since it efficiently exploits spatial locality. However, it is often beneficial to fetch a cache block that has a certain offset to the current block since the processor is much faster than main memory. This parameter is called the prefetch *distance*. Furthermore, it is possible to fetch more than one block at the time. This is known as the prefetch *degree*. A *tagged sequential* prefetcher improves on the sequential prefetcher by adding a bit to each cache line that is set for prefetched blocks [148]. If the process then hits on a prefetched block, the prefetcher knows that the prefetch was successful and can initiate a request for the next line.

A sequential prefetcher will fetch unnecessary data if the process does not access memory in a continuous fashion. *Reference Prediction Table (RPT)* prefetching improves on this by storing the distance between accesses [19, 22]. Figure 2.8 illustrates the structure of an RPT table entry. In this method, each table entry is indexed by the *Program Counter (PC)* value of the load instruction. On the first miss with a particular PC value, a 4-state RPT prefetcher typically stores the memory address in the *Last Address* field. On the second miss, the difference between the current address and the last address is computed and stored in the *Stride* field. This address difference is often referred to as a *delta*. In addition, the *Last Address* field is updated. On the third miss, the prefetcher starts prefetching if the address difference is equal to the stored stride.

RPT is limited to handling constant strides, but more complex access patterns can occur. Consider the access example in Figure 2.10. To handle such access patterns, Nesbit and Smith proposed the *Global History Buffer (GHB)* which is shown in Figure 2.9 [105, 107]. The GHB is a fixed-length FIFO table which stores the addresses of the most recent cache misses and hits to previously prefetched cache

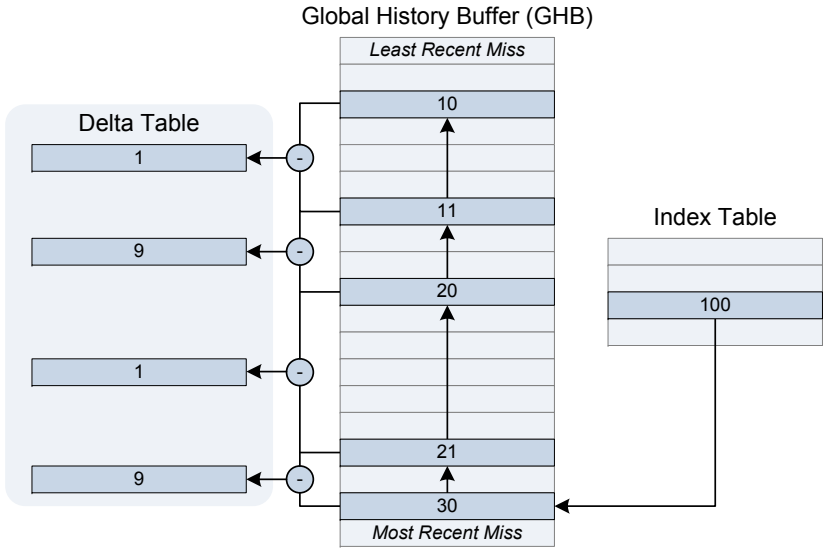


Figure 2.9: Delta Table Construction Example

Address	10		11		20		21		30
Delta		*1*		*9*		*1*		*9*	

Figure 2.10: Memory Access Example

blocks. The cache misses are aggregated into groups and each miss address contains a pointer to the previous miss address within that group. For instance, the cache misses are grouped with the PC address of the load in *Program Counter/Delta Correlation (PC/DC)* prefetching. Figure 2.9 shows the contents of the GHB for the access example in Figure 2.10.

PC/DC prefetching requires the address of the load. Consequently, either the address of the load must be transmitted alongside the miss address or the prefetcher must be coupled to the processor core. An alternative solution is to do prefetching based on the miss address. *CZone/Delta Correlation (C/DC)* prefetching is a variant of PC/DC that divides memory into fixed-size concentration zones (CZones) [108]. Then, the CZones are used instead of the PC to group requests. The grouping of requests into streams is called *localization* [26]. Diaz and Cintra [26] observed that timeliness could be increased by partially reconstructing the temporal order of accesses to different streams. They implemented this scheme by adding a pointer to each GHB index table element that points to the likely temporal successor.

PC/DC and C/DC selects prefetches with *Delta Correlation (DC)*. First, the GHB linked list for the PC or CZone is traversed to produce a sequence of deltas. This

sequence is stored in the *Delta Table*. DC is the process of searching for the two most recent deltas in this delta history. In Figure 2.10, the two most recent deltas (1 and 9) match the oldest pair of deltas. The delta after the match (1) is then used to calculate the next address (31), and a prefetch for this address is issued.

The main GHB drawback is the complexity associated with retrieving the delta history. Grannæs et al. [44] provides *Delta Correlating Prediction Tables (DCPT)* prefetching which can detect the same patterns as GHB but with a simpler table-based implementation inspired by RPT. The key difference between DCPT and RPT is that each table entry can hold multiple deltas.

Finally, Lin et al. [89] used *Scheduled Region Prefetching (SRP)* to bridge the performance gap between the processor and DRAM. Here, they divide memory into fixed-size regions. If the needed memory bus channel is idle, the whole region is prefetched which can result in very high memory bus utilization.

2.4.2 Memory Controller Prefetch Scheduling

A central decision when integrating the prefetcher with the memory controller is the relative priorities of prefetches and demand accesses. The two simplest policies are to either prioritize demand reads over prefetches or prioritize demand reads and prefetches equally. Lee et al. [85] observed that it is beneficial with equal priorities when prefetcher accuracy is high while prioritizing reads are useful when prefetcher accuracy is low. Since the accuracy of the prefetcher depends on the characteristics of the running process, Lee et al. provide an adaptive scheme that chooses the relative priorities based on dynamic accuracy measurements.

2.4.3 Prefetching in CMPs

Each core may have its own prefetcher in CMPs. Allowing these prefetchers to operate in an uncontrolled fashion may create significant interference with both demand and prefetch accesses of other cores. Ebrahimi et al. [31] provides a scheme for reducing prefetcher-caused interference by throttling the prefetchers depending on the state of the memory system. This throttling is based on both local and global feedback. The global feedback is used to avoid making decisions that are good from a local point of view but will reduce system performance.

Lee et al. [86] provides a different way of integrating prefetchers in a CMP with no shared cache. Here, they choose prefetch requests to maximize *Memory Level Parallelism (MLP)*. In modern DRAM systems, the amount of MLP is closely tied to how efficiently the process is able to utilize the available DRAM banks. Consequently, Lee et al. chooses prefetches to maximize the bank parallelism of the concurrent requests. Furthermore, they load the concurrent requests of one processor into the memory bus queue at the same time to minimize request serialization.

Chapter 3

Methodology

This chapter discusses the experiment methodology used in the papers included in this thesis. This methodology is used to quantify the effects of our contributions. Section 3.1 explains why a simulator-based methodology is used and the reasons for choosing the M5 simulator [11]. Then, benchmarks are discussed is Section 3.2. In Section 3.3, multiprogrammed workload generation is discussed. In addition, this section contains an analysis of how accurate multiprogrammed metric results can be provided. Finally, our use of compute clusters for *Design Space Exploration (DSE)* is discussed in Section 3.4.

3.1 Simulators

Computer architectures can be evaluated in three main ways [129]:

- Performance measurement on real hardware
- Simulation
- Analytical modeling

In this thesis, we investigate new hardware techniques. Unfortunately, a significant effort is involved in implementing these techniques in real hardware. A simulator-based approach is more efficient since it enables rapid iterations through the improvement and evaluation loop. Furthermore, modern simulators have a sufficient level of detail to make the effects our techniques aim to alleviate observable. Analytical modeling has a significant advantage for exploration of large design spaces and early studies of future technologies that are very different from current simulation models [129]. Since our research focuses on architectures that are similar to current CMPs, the current simulators serve our purpose.

There is no shortage of computer architecture simulators. Therefore, finding the most suitable simulator can be a challenging task. Previously, our research group have used the SimpleScalar simulator [7, 27]. Unfortunately, SimpleScalar does not support simulating CMPs without modifications. Furthermore, memory latencies are calculated in a single operation which makes it difficult to model request interleaving and queuing effects. Consequently, we started to look for a SimpleScalar replacement. An important step in this process was Lande's master thesis [84] where he evaluated Rsim [57], Asim [33], SimOS [121], Simics [95], TFSim [98], SimFlex [49], GEMS [97] and M5 [11]. Then, he carried out a thorough evaluation of M5 to establish if it met the needs of the research group. In the end, we decided to use M5 since it offered CMP support and an event-driven memory hierarchy. An event-driven memory hierarchy makes it possible to accurately model queuing and interleaving of memory requests which is a central theme in this thesis.

M5 is an *execution-driven* simulator which makes it possible to capture dynamic interactions between instructions and memory requests. In an execution-driven simulator, a benchmark binary is used to drive the simulated CMP. Alternatively, a *trace-driven* simulator uses a trace of the executed instructions or memory requests to drive the simulator model. Furthermore, M5 supports both *system call emulation* and *full-system* simulation. With full-system simulation, the simulator runs an *Operating System (OS)*. In contrast, all system calls are handled by the host OS with system call emulation. Although full-system simulation is more realistic, it also makes it difficult to find the cause of the observed behavior. Therefore, we use system call emulation in this thesis and leave full-system evaluation as further work.

Choosing system call emulation makes running multi-threaded benchmarks complicated. The reason is that communication libraries often have significant interaction with the OS. Consequently, it is likely that a large number of system calls would need to be implemented. Full-system simulation avoids this problem because the simulated OS provides these features. System call emulation also makes it challenging to adopt new benchmarks and compilers since they often require new system calls.

Although M5 was well suited to the needs of the research group, we had to implement significant extensions for it to fit our needs. Firstly, we have replaced the on-chip bus model with a range of different interconnect topologies. Secondly, the simple off-chip memory bus model has been replaced by a detailed DDR2 model and various memory bus schedulers (FCFS, FR-FCFS [120] and NFQ [119]). Thirdly, we have implemented multi-banked shared caches, an *Auxiliary Tag Directory (ATD)* [30, 117] and MTP cache partitioning [18]. Fourthly, we have extended M5 to collect basic block vectors for SimPoints [48] as well as improving the checkpointing support. Finally, we have developed a large number of Python scripts that help us run our experiment and analyze the results.

3.2 Benchmarks

3.2.1 Choosing Benchmarks

In this thesis, we use the SPEC CPU2000 benchmark suite [136] which provides single-threaded benchmarks. As technology evolves, the benchmark suite requirements change. Consequently, SPEC CPU2000 was retired in 2007 and replaced by SPEC CPU2006 [51]. Unfortunately, supporting new benchmarks in system call emulation is not trivial. In fact, even the newest version of M5 does not support the full SPEC CPU2006 suite at the time of writing [94]. Therefore, we did not prioritize supporting CPU2006 for this work.

A program must be multi-threaded to use all CMP resources to solve a single task. Consequently, it is likely that this class of programs will become more important in the future, and a number of benchmark suites are available [9, 128]. In this thesis, we limit the study to multiprogrammed workloads that are collections of single-threaded benchmarks. Lifting this restrictions and studying the performance of workloads with both single-treaded and multi-threaded benchmarks is left as future work.

3.2.2 Representative Benchmark Simulation

The SPEC CPU2000 benchmarks are real programs chosen to stress the processor and memory system of real systems. Consequently, simulating them in detail to completion may take weeks or even months. This problem can be avoided by finding parts of the benchmarks that are small enough to simulate while still representing the behavior of the complete benchmark. The start of a benchmark is unlikely to be representative since it often initializes data structures and carry out other administrative tasks. For this reason, it is common to functionally emulate benchmark execution up to a certain point which is known as *fast-forwarding*. Then, simulation statistics are reset and detailed simulation is started. The advantages of this method are that it is simple and easy to apply to CMP systems. Therefore, we use clock cycle fast-forwarding in most of the papers in this thesis. However, it is unlikely that the chosen interval is representative of the complete benchmark.

For this reason, researchers have proposed different methods for choosing representative parts of benchmarks. The SimPoint methodology provides representative benchmark samples by profiling the execution frequency of each basic block in each sample [48, 114]. Then, a clustering algorithm is used to group similar samples, and each group is assigned a weight depending on how many of the samples this cluster represents. These weights make it possible to combine the performance measurements from each sample into a value that is representative for the full benchmark. Wunderlich et al. [152] use statistical sampling to achieve the same effect with their *Sampling Microarchitecture Simulation (SMARTS)* methodology. Here, the user supplies the desired confidence and SMARTS computes the number

of samples needed to reach this level of confidence. The accuracy of SimPoints and SMARTS has been validated in an independent study [155]. We use SimPoints in the last paper in this thesis (Appendix C).

3.3 Simulating Multiprogrammed Workloads

We create multiprogrammed workloads by choosing benchmarks at random from the whole SPEC CPU2000 benchmark suite. Furthermore, we sometimes limit the benchmarks available to the selection process to make certain behaviors more likely (see Appendix A). We avoid that two instances of the same benchmark start on the same instruction in the detailed simulation since this is a very uncommon event in real systems and may result in severe interference. To avoid this problem, we either fast-forward the instances by a different amount or only allow a benchmark to be used once in a workload.

Vera et al. [149] presents one way of providing representative performance metric measurements from multiprogrammed workloads called FAME. They observed that the average IPC for each benchmark will converge if the benchmark samples are re-executed. Furthermore, they provide an analytical expression that calculates the number of iterations necessary to put a bound on the IPC variance. FAME relies on a single simulation point for each benchmark which can be fairly representative of full benchmark execution [10]. However, this point is only one out of possibly many benchmark phases which may lead to interesting behavior being missed. Co-Phase Matrix simulation [10] supports multiple simulation points for each benchmark, but it does not scale to a large number of cores [149]. In addition, FAME retrieves simulation statistics when the last benchmark has reached its required number of iterations. Consequently, the number of instructions executed by the other benchmarks is not bounded. To ensure that private and shared mode statistics are computed based on the same instructions, it may be necessary with a private mode run for each benchmark instance.

For these reasons, we do not use the FAME methodology in our SimPoint based experiments. Instead, we use a method inspired by Xie and Loh [153] where we retrieve statistics when a benchmark reaches the required number of instructions, but continue running until all benchmarks have committed enough instructions. With this method, private to shared mode comparisons are straightforward since all results contain the same instructions. In contrast with Xie and Loh [153], we use multiple simulation points, and Figure 3.1 illustrates our method. First, we assign an identifier from 1 to s to each of the simulation points of a benchmark. Then, we group all simulation points with the same IDs into workloads and run these workloads in the simulator. Then, we retrieve the results from each core and create an aggregate value with the weights provided by the SimPoint tool. We use the function $w(b, s)$ to identify the weights in Figure 3.1. In this function, b is the benchmark and s is the simulation point identifier.

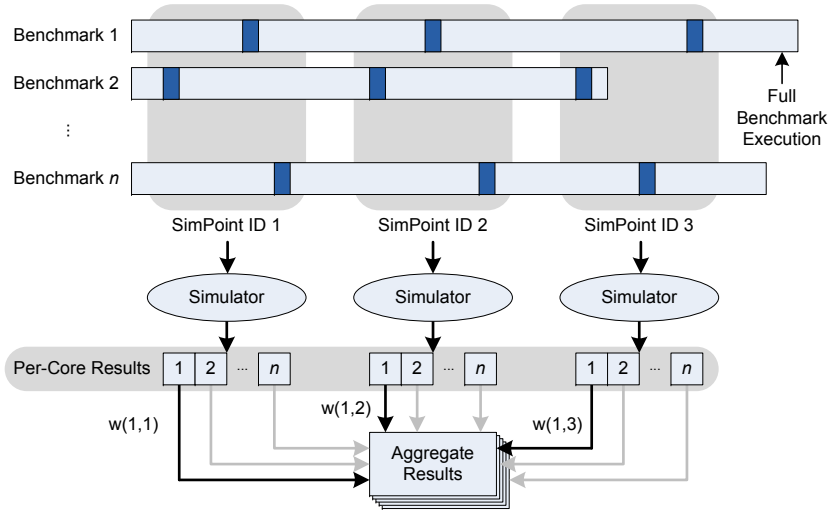


Figure 3.1: SimPoint-Based Multiprogrammed Workload Simulation Methodology

Due to heavy interference, it can take up to 7 billion clock cycles for a benchmark to commit 25 million instructions in our experiments (see Appendix C). By using multiple simulation points, we can provide results that represent a large number of instructions from smaller samples. Since these workloads are independent of each other, they can be run in parallel. In this way, we trade simulation latency for simulation bandwidth which significantly increases the efficiency of the method. To improve performance further, we also create a checkpoint for each simulation point which eliminates the need for fast-forwarding. This checkpoint contains cache state which reduces the need for warm-up.

3.4 Design Space Exploration

Most computer architecture research contains an element of *Design Space Exploration (DSE)*. Commonly, a number of architectural configurations are examined with different workloads or benchmarks, and each combination of workload and architectural configuration becomes a point in the design space. In practice, these points are created with different simulator command lines. Commonly, the design spaces are large, and it is necessary to write software that runs the experiments. Furthermore, large experiments provide large amounts of data which creates the need for software that retrieves statistics and visualizes results.

Each point in the design space is independent of all other points. Consequently, they can be evaluated in parallel which makes computer architecture research well

suited to large clusters. Fortunately, the *Norwegian Metacenter for Computational Science (NOTUR)* [111] has consistently provided us with annual allocations of 500000 to 600000 CPU hours (57 to 68 CPU years) on the 5632-core Stallo cluster [147]. This has been a great advantage since it has enabled us to thoroughly evaluate our proposed techniques.

Chapter 4

Research Process

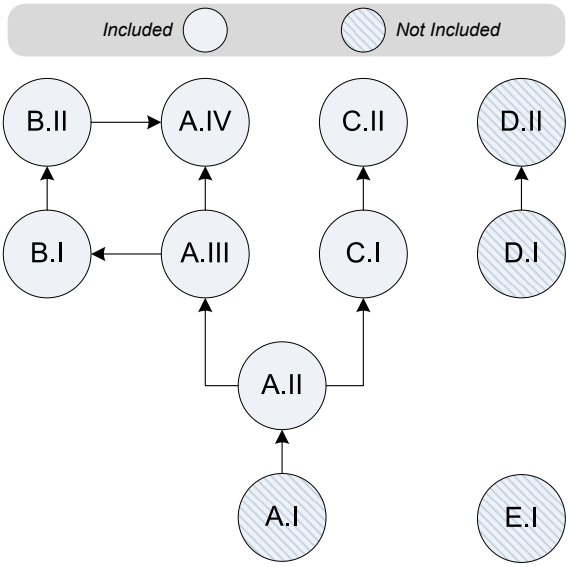
This chapter discusses how the papers in this thesis were produced. Figure 4.1 views this process from two different angles. Firstly, Figure 4.1(a) illustrates the logical structure of the papers and how they are related. Secondly, Figure 4.1(b) shows the duration and concurrency of the work with the different papers. In this figure, a filled box indicates that the paper is actively worked on and a transparent box illustrates that the article is under review or being prepared for publication. To simplify the discussion, the papers are grouped into five categories according to the main contribution of each paper. The categories are summarized in Table 4.1.

4.1 Preliminary Work

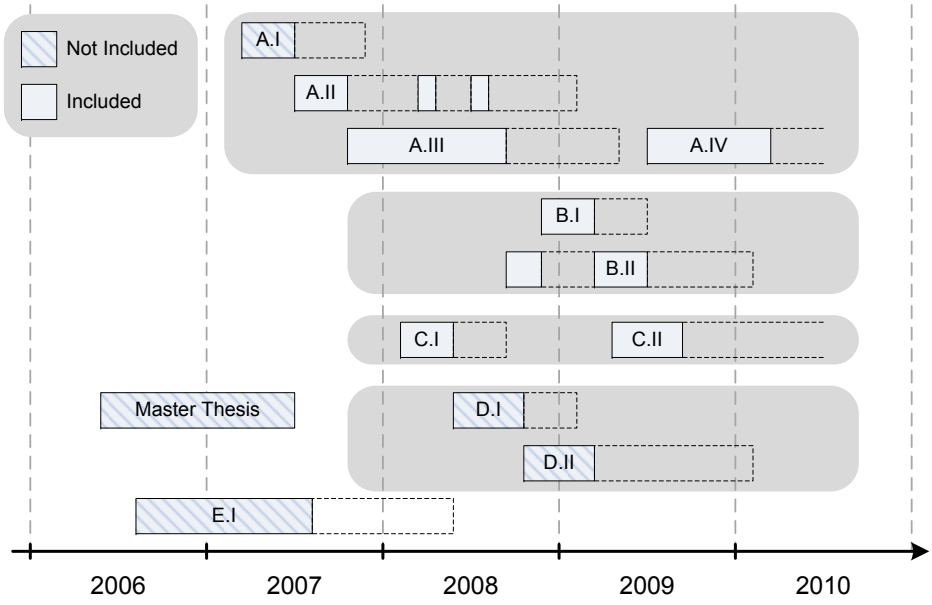
The work described in this thesis has been carried out within the Integrated PhD Program at the IME faculty. In this program, the last year of the master study overlaps with the start of the PhD. To take advantage of the synergies of concurrently working towards two degrees, the master thesis topic is chosen to be a suitable starting point for future research. Originally, we planned to investigate the

Table 4.1: Paper Categories

Category	**Name**	**Papers**	
		Total	*Included*
A	Adaptive Miss Handling Architectures	4	3
B	Memory System Interference	2	2
C	CMP Prefetch Scheduling	2	2
D	Prefetching Systems	2	0
E	Learning and ICT	1	0



(a) Logical Structure



(b) Chronology

Figure 4.1: Paper Overview

Table 4.2: Paper Category A

ID	Title	Ref.
A.I	Performance Effects of a Cache Miss Handling Architecture in a Multi-core Processor	[66]
A.II	A High Performance Adaptive Miss Handling Architecture for Chip Multiprocessors	[67]
A.III	A Light-Weight Fairness Mechanism for Chip Multiprocessor Memory Systems	[68]
A.IV	Managing Chip Multiprocessor Memory Systems with Miss Bandwidth Allocations	-

performance of parallel applications on a CMP, and this was the topic of my master thesis [65]. This avenue was not pursued further because tool support for such research topics were lacking at the time. Furthermore, I concurrently discovered the *Dynamic Miss Handling Architecture (DMHA)* mechanism that is the basis for paper Category A and provides a use case for the contributions in Category B.

4.2 Category A: Adaptive Miss Handling Architectures

The Category A papers (see Table 4.2) originated from my effort to try to create a realistic processor model for the experiments in my master thesis. In particular, it turned out to be difficult to choose a realistic number of MSHRs for the caches. Since I had to write a term project for a PhD-level computer architecture course at that time, I decided to investigate the performance impact of the number of MSHRs in a CMP. In this work, I observed that too many MSHRs can create memory bus congestion and degrade performance. This term project became Paper A.I which is not included in this thesis since its main results are covered by subsequent contributions.

The next step was to use this observation to improve performance. It turned out that while too much miss bandwidth degrades performance in some cases, too little miss bandwidth degrades performance in other cases. Therefore, we developed an adaptive technique that tries to achieve the best of both worlds. We observed that off-chip bandwidth was the main bottleneck, and used this to guide bandwidth allocations. Our first cut at this task eventually became the *Adaptive Miss Handling Architecture (AMHA)* and Paper A.II. However, getting this paper published turned out to be a challenge. As expected, our first submission to ISCA resulted in heavy criticism and rejection. Since the main part of the criticism was aimed at our simplistic memory bus and DRAM model, we decided to develop significantly more detailed memory bus and DRAM model. This model is used in

all publications included in this thesis, and the contributions in Category C would not have been possible without it.

Our next step was to submit the paper to the HiPEAC Journal. However, we quickly discovered that we used a subset of the cache index bits for bank selection in our multi-banked caches. Consequently, only $\frac{1}{b}$ of the cache capacity was used with b banks. The journal editor allowed us to fix this problem and resubmit the paper. Again, the paper was criticized by the reviewers, especially for our use of the *Aggregate IPC (AI)* metric. However, the editor decided that the paper would be accepted if we improved our evaluation. The improved evaluation made it necessary to modify the technique, but the idea remained the same. Although this process was challenging at the time, it significantly increased the quality of the paper. Furthermore, the improvements to the methodology have been of great benefit to later contributions.

While working on Paper A.II, I read a number of papers on CMP resource sharing and became curious about whether an adaptive MHA could be used to provide fairness. We coined the term *Dynamic Miss Handling Architecture (DMHA)* for a MHA where the number of MSHRs can be changed at runtime. As a first step towards achieving fairness with a DMHA, I devised an experiment that measures the performance of a large number of static, asymmetric MHAs. Paper A.III was built around this experiment, and established that the DMHA mechanism could be used to improve both performance and fairness in CMP memory systems. In Paper A.III, we also tried to guide the DMHA with simple interference measurements. We called this system the *Fair Adaptive Miss Handling Architecture (FAMHA)*. Unfortunately, FAMHA was only able to achieve the potential of the static asymmetric MHAs in a few cases. To improve this system, we decided to investigate how accurate interference measurements could be provided. This turned out to be more challenging than first expected and resulted in the contributions in Category B.

Paper A.IV leverages the preceding contributions in categories A and B to manage miss bandwidth allocations in CMP memory systems. In Paper A.IV, we introduce *Miss Handling Architecture Bandwidth Control (MHABC)* which models the latency effects of miss bandwidth allocations to dynamically manage the DMHAs. MHABC can optimize for a range of system performance metrics. Furthermore, the MHABC approach is general and can be applied if the latency cost of allocation decisions can be modeled with sufficient accuracy. At the time of writing, Paper A.IV has been submitted to IEEE Transactions on Computers.

4.3 Category B: Memory System Interference

Although Paper A.III provided promising results for the DMHA mechanism, it did not provide a practical implementation. To provide such a system, we decided to develop an accurate interference feedback mechanism. An important design goal was that we should be able to quantify the measurement error of the mechanism.

Table 4.3: Paper Category B

ID	Title	Ref.
B.I	A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures	[69]
B.II	DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems	[70]

Since this goal was difficult to achieve with the *Interference Point (IP)* mechanism from Paper A.III, we decided to try a different approach.

The main inspiration for this work was Mutlu and Moscibroda [102] and their definition that interference is the additional time the processor is stalled waiting for memory in the shared mode. Processor stall time due to memory depends on the ability of the process to tolerate memory latencies and utilize the parallelism of the memory system. Consequently, it is difficult to measure this quantity directly. Instead, we decided to define interference in terms of the the additional memory latency which can be measured directly. Furthermore, the total latency is the sum of the latency in each memory system unit which makes the units' measurement techniques relatively independent.

We started working on the system that would eventually become the main contribution of Paper B.II, in the autumn of 2008. Although we quickly developed a prototype, its accuracy was poor and it proved difficult to find the cause of the inaccuracies. By December, it was clear that we needed to improve our understanding of the problem. To achieve this, we started to work on Paper B.I. In Paper B.I, we quantified the latency impact of each shared unit by comparing the latency of each memory request in the private and shared modes. As well as improving our understanding, this work helped us achieve synchronized measurements of shared and private mode memory requests.

When Paper B.I was finished, we continued work on Paper B.II. This time, progress was better and we managed to track down the major problems. These problems were either programming errors or related to combination effects between the shared cache and memory bus. In particular, shared cache writebacks occur at different points in the benchmarks execution in the private and shared modes. We finished Paper B.II in the beginning of summer 2009.

4.4 Category C: CMP Prefetch Scheduling

The feedback we got on the first submission of Paper A.II made it clear that the memory bus and DRAM model of the M5 simulator was too simplistic for our research topics. Concurrently, my fellow PhD student Marius Grannæs was porting his prefetcher implementations from SimpleScalar to M5. Grannæs had

Table 4.4: Paper Category C

ID	Title	Ref.
C.I	Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors	[43]
C.II	Exploring the Prefetcher/Memory Controller Design Space: An Opportunistic Prefetch Scheduling Strategy	-

already observed that the benefits of prefetching are closely tied to DRAM page locality [41]. Consequently, we decided to join forces and develop a detailed memory model based on the DDR2 standard document [71].

During the implementation of this model, we became interested in memory access scheduling. We started by implementing a simple *First Come First Served (FCFS)* scheduler and the *First Ready - First Come First Served (FR-FCFS)* scheduler by Rixner et al. [120]. While porting his prefetcher implementations to M5, Grannæs observed that cleverly scheduling prefetches and demand reads can improve performance. In Paper C.I, we *piggybacked* prefetches to open pages on demand reads to these pages which make prefetches cheaper than ordinary reads. We observed that prefetching improved performance as long as the accuracy of the prefetcher was above 38%. This threshold is found empirically and indicates the break-even point between the cost of prefetching and the cost of regular reads. In other words, it indicates the amount of useless data we can allow the prefetcher to fetch from open pages without degrading performance.

Paper C.II was born as an idea for a new prefetching heuristic. Grannæs observed that the state of the DRAM system could be used to generate prefetches that can be efficiently executed. This is the opposite approach to conventional prefetching heuristics which create prefetches based on the miss address stream. The key component of this system is the *Page Vector Table (PVT)* which contains one bit for each cache line in a DRAM page. While working on this idea, we realized that the PVT could be used as the interface between the prefetcher and the memory bus scheduler. In this system, the prefetcher sets the bits of the cache lines it wants to retrieve in the PVT which facilitate efficient prefetch scheduling. In our *opportunistic* prefetch scheduling strategy, we fetch all marked cache blocks in the PVT at the time the memory bus scheduler closes the page if the accuracy of the prefetcher is sufficiently high. Paper C.II also explores the prefetch scheduling design space, indicating that the opportunistic strategy has an advantage when bandwidth constrained CMPs are combined with aggressive prefetchers. At the time of writing, Paper C.II is being reviewed by the Journal of Computer Science and Technology.

In this thesis, we investigate resource management in CMP memory systems. While the contributions in categories A manage off-chip bandwidth to improve system-wide performance metrics, prefetching aims to put the available bandwidth to good use. Consequently, it provides more bandwidth to processes that have predictable memory access patterns. This may decrease the performance of processes with

Table 4.5: Paper Category D

ID	Title	Ref.
D.I	Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables	[44]
D.II	Multi-level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matching	[45]

access patterns that are difficult to predict. In both Paper C.I and C.II, we show that our prefetch scheduling techniques reduce the maximum slowdown relative to no prefetching. The reason is that prefetching to open pages in many cases can be carried out in cycles that would otherwise be wasted due to DRAM command dependencies.

4.5 Category D: Prefetching Systems

Perez et al. [115] compared a large number of data prefetching heuristics and found that prefetcher performance could be very different depending on benchmark selection, choice of simulator and other methodological considerations. Furthermore, they found that the published papers often lacked significant implementation details. For these reasons, they proposed that the research community should use a common simulator infrastructure to facilitate efficient sharing of the implementation of the new techniques.

To address these issues, the *Journal of Instruction Level Parallelism (JILP)* organized the first *Data Prefetching Championship (DPC)* early in 2009. The organizers provided a common simulator infrastructure with a simple programming interface to ensure fair and accurate comparison of the different prefetchers. Since Grannæs already had considerable experience with many different prefetchers, we decided to participate in the championship. Our prefetcher was able to detect the same patterns as the top performing PC/DC prefetcher [107] in the study of Perez et al. with a lower storage cost. We entered the competition with Paper D.I and finished in fourth place. We further observed that the key ideas from the three better performing prefetchers could easily be incorporated into our prefetching system. This observation resulted in Paper D.II.

Category D is not included in this thesis because it focuses on prefetching in a single-core processor. Consequently, the task is to achieve the best possible utilization of the memory system for a single process. Although this problem is similar to the core issues in this thesis, it is not the same. Therefore, Category D is left out to keep a clear focus on CMP memory systems.

Table 4.6: Paper Category E

ID	Title	Ref.
E.I	Experimental Validation of the Learning Effect for a Pedagogical Game on Computer Fundamentals	[127]

4.6 Category E: Learning and ICT

My involvement with Paper E.I was a result of my duty work as a teaching assistant in our computer fundamentals course. In this course, the main task of the teaching assistant is the day to day operation of the *Age of Computers (AoC)* educational game. AoC also contributes to *Learning and ICT (LICT)* which is selected as a strategic research area for NTNU. In the autumn of 2006, Sindre and Natvig decided to carry out an experiment to investigate the learning effect of AoC since this had been requested by a reviewer on a pending journal article. Since I was running the AoC system at the time, I was asked to prepare a version of AoC for the experiment as well as assist with other practical matters. This experiment eventually became Paper E.I.

Chapter 5

Research Results

The aim of this chapter is to provide an overview of the papers included in this thesis. The included papers are discussed in sections 5.1 through 5.7. These sections contain the abstract of the paper and a description of the contributions of the different co-authors. Most of the sections also contain a discussion on how the paper is viewed in retrospective. Paper A.IV, Paper B.II and Paper C.II were finished very recently so the sections describing these papers do not include a retrospective view. Finally, Section 5.8 lists the papers that were not included in this thesis.

5.1 Paper A.II

A High Performance Adaptive Miss Handling Architecture for Chip Multiprocessors
M. Jahre and L. Natvig
Transactions on High-Performance Embedded Architectures and Compilers
2009

5.1.1 Abstract

Chip Multiprocessors (CMPs) mainly base their performance gains on exploiting thread-level parallelism. Consequently, powerful memory systems are needed to support an increasing number of concurrent threads. Conventional CMP memory systems do not account for thread interference which can result in reduced overall system performance. Therefore, conventional high bandwidth Miss Handling Architectures (MHAs) are not well suited to CMPs because they can create severe

memory bus congestion. However, high miss bandwidth is desirable when sufficient bus bandwidth is available. This paper presents a novel, CMP-specific technique called the Adaptive Miss Handling Architecture (AMHA). If the memory bus is congested, AMHA improves performance by dynamically reducing the maximum allowed number of concurrent L1 cache misses of a processor core if this creates a significant speedup for the other processors. Compared to a 16-wide conventional MHA, AMHA improves performance by 12% on average for one of the workload collections used in this work.

5.1.2 Roles of the Authors

I did most of the work on this paper. Natvig worked as an advisor and provided a large number of helpful comments and improvements to the revisions of this paper.

5.1.3 Retrospective View

In this paper, we use the *Multiprogrammed Baseline (MPB)* as the basis for computing the system performance metrics. In other words, the baseline configuration has a static and equal partitioning of memory bus bandwidth and cache space. This is not ideal since it makes it difficult to know if the performance improvements are due to the AMHA technique or due to the process being able to grab more cache space in the shared mode. Furthermore, *Network Fair Queuing (NFQ)* partitions bandwidth but does not necessarily latency [102, 119]. However, the main performance trends are consistent with Paper A.IV which uses a significantly improved methodology.

5.2 Paper A.III

A Light-Weight Fairness Mechanism for Chip Multiprocessor Memory Systems
M. Jahre and L. Natvig
ACM International Conference on Computing Frontiers
2009

5.2.1 Abstract

Chip Multiprocessor (CMP) memory systems suffer from the effects of destructive thread interference. This interference reduces performance predictability because it depends heavily on the memory access pattern and intensity of the co-scheduled threads. In this work, we confirm that all shared units must be thread-aware

in order to provide memory system fairness. However, the current proposals for fair memory systems are complex as they require an interference measurement mechanism and a fairness enforcement policy for all hardware-controlled shared units. Furthermore, they often sacrifice system throughput to reach their fairness goals which is not desirable in all systems.

In this work, we show that our novel fairness mechanism, called the Dynamic Miss Handling Architecture (DMHA), is able to reduce implementation complexity by using a single fairness enforcement policy for the complete hardware-managed shared memory system. Specifically, it controls the total miss bandwidth available to each thread by dynamically manipulating the number of Miss Status Holding Registers (MSHRs) available in each private data cache. When fairness is chosen as the metric of interest and we compare to a state-of-the-art fairness-aware memory system, DMHA improves fairness by 26% on average with the single program baseline. With a different configuration, DMHA improves throughput by 13% on average compared to a conventional memory system.

5.2.2 Roles of the Authors

The roles of the authors were accurately captured by the description in Section 5.1.2.

5.2.3 Retrospective View

In this paper, we use both the *Multiprogrammed Baseline (MPB)* and the *Single Program Baseline (SPB)*. Consequently, it is an improvement compared to Paper A.II, but there is still some way to go. In particular, fast forwarding for a fixed number of clock cycles may result in a different part of the benchmark being used in the private and shared modes. If these parts are from different benchmark phases, the results will be strange. We cannot see any evidence of this situation in the results in the paper, but at the same time we cannot say that it did not occur. The most important result in this publication is the results from the *offline-best-static* configuration, and these results are consistent with the results in Paper A.IV.

5.3 Paper A.IV

Managing Chip Multiprocessor Memory Systems with Miss Bandwidth Allocations
M. Jahre, M. Grannæs and L. Natvig
Submitted to: IEEE Transactions on Computers
2010

5.3.1 Abstract

Chip Multiprocessors (CMPs) share on-chip units to achieve good resource utilization. This design choice makes destructive interference possible and may cause performance degradations. Resource allocation systems can avoid this problem, and previous approaches have provided separate resource allocation systems for each shared hardware-controlled unit. In this work, we show that resource sharing can be globally controlled by carefully orchestrating the miss bandwidth available to each running process. Our Miss Handling Architecture Bandwidth Control (MHABC) technique improves system performance by tuning the maximum number of concurrent shared memory requests for each process to runtime interference patterns. MHABC leverages a novel interference measurement methodology that estimates the interference-free IPC of a process with an average error of -0.3% and a standard deviation of 12.0%. When MHABC is configured to optimize for the Harmonic Mean of Speedups (HMoS) metric, it improves HMoS by up to 106% and fairness by up to 200% with a worst-case reduction in throughput of 3%.

5.3.2 Roles of the Authors

I did most of the work on this paper. However, I did have several interesting discussions with Grannæs in the early phases of this work, and he provided many helpful comments on the paper. Furthermore, Grannæs had the original idea for how a set-sampled *Auxiliary Tag Directory (ATD)* could be integrated with the memory bus interference technique. Natvig worked as an advisor and provided a large number of helpful comments and improvements.

5.4 Paper B.I

A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures
M. Jahre, M. Grannæs and L. Natvig
11th IEEE International Conference on High Performance Computing and Communications
2009

5.4.1 Abstract

The potential for destructive interference between running processes is increased as Chip Multiprocessors (CMPs) share more on-chip resources. We believe that understanding the nature of memory system interference is vital to achieve good fairness/complexity/performance trade-offs in CMPs. Our goal in this work is to quantify the latency penalties due to interference in all hardware-controlled, shared units (i.e. the on-chip interconnect, shared cache and memory bus). To achieve this, we simulate a wide variety of realistic CMP architectures. In particular, we vary the number of cores, interconnect topology, shared cache size and off-chip memory bandwidth. We observe that interference in the off-chip memory bus accounts for between 63% and 87% of the total interference impact while the impact of cache capacity interference can be lower than indicated by previous studies (between 5% and 32% of the total impact). In addition, as much as 11% of the total impact can be due to uncontrolled allocation of shared cache Miss Status Holding Registers (MSHRs).

5.4.2 Roles of the Authors

I had the initial idea, carried out the preliminary investigations and proposed the initial design. This design was then refined through extensive discussions with Grannæs. I implemented the refined idea in the common simulator framework which contains code produced by both Grannæs and myself. Furthermore, I devised the initial experimental methodology and planned which experiments should be carried out. The experiment plan and methodology was then discussed thoroughly with Grannæs.

I wrote the first draft and had the final word in all matters regarding the paper. Grannæs read the draft thoroughly and provided significant improvements to the presentation, organization and language. Natvig helped with proof-reading and guidance.

5.4.3 Retrospective View

In this paper, we observed that the latency cost of cache capacity interference can be a small part of the total interference latency. However, it is not clear from the paper that cache interference is the only part of the total interference latency that a resource management technique may remove completely. Consequently, the results should not be taken as evidence that shared cache resource management is not useful. However, we do advocate that shared cache techniques should be evaluated with a realistic bus model to give a more correct view of their overall impact.

5.5 Paper B.II

DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems
M. Jahre, M. Grannæs and L. Natvig
5th International Conference on High Performance and Embedded Architectures and Compilers
2010

5.5.1 Abstract

Chip Multi-Processors (CMPs) commonly share hardware-controlled on-chip units that are unaware that memory requests are issued by independent processors. Consequently, the resources a process receives will vary depending on the behavior of the processes it is co-scheduled with. Resource allocation techniques can avoid this problem if they are provided with an accurate interference estimate. Our Dynamic Interference Estimation Framework (DIEF) provides this service by dynamically estimating the latency a process would experience with exclusive access to all hardware-controlled, shared resources. Since the total interference latency is the sum of the interference latency in each shared unit, the system designer can choose estimation techniques to achieve the desired accuracy/complexity trade-off. In this work, we provide high-accuracy estimation techniques for the on-chip interconnect, shared cache and memory bus. This DIEF implementation has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

5.5.2 Roles of the Authors

The roles of the authors were accurately captured by the description in Section 5.4.2.

5.6 Paper C.I

Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors
M. Grannæs, M. Jahre and L. Natvig
XXVI IEEE International Conference on Computer Design
2008

5.6.1 Abstract

The pressure on off-chip memory increases significantly as more cores compete for the same resources. A CMP deals with the memory wall by exploiting thread level parallelism (TLP), shifting the focus from reducing overall memory latency to memory throughput. This extends to the memory controller where the 3D structure of modern DRAM is exploited to increase throughput.

Traditionally, prefetching reduces latency by fetching data before it is needed. In this paper we explore how prefetching can be used to increase memory throughput. We present our own low-cost open-page prefetch scheduler that exploits the 3D structure of DRAM when issuing prefetches. We show that because of the complex structure of modern DRAM, prefetches can be made cheaper than ordinary reads, thus making prefetching beneficial even when prefetcher accuracy is low. As a result, prefetching with good coverage is more important than high accuracy. By exploiting this observation our low-cost open page scheme increases performance and QoS. Furthermore, we explore how prefetches should be scheduled in a state of the art memory controller by examining sequential, scheduled region, CZone/Delta Correlation and reference prediction table prefetchers.

5.6.2 Roles of the Authors

Grannæs had the initial idea, carried out the preliminary investigations and proposed the initial design. This design was then refined through extensive discussions with me. Grannæs implemented the refined idea in the common simulator framework which contains code produced by both Grannæs and myself. Furthermore, Grannæs devised the initial experimental methodology and planned which experiments should be carried out. The experiment plan and methodology was then discussed thoroughly with me.

Grannæs wrote the first draft and had the final word in all matters regarding the paper. I read the draft thoroughly and provided significant improvements to the presentation, organization and language. Natvig helped with proof-reading and guidance.

5.6.3 Retrospective View

For this paper, a clarification about the aggregated performance measurements is in order. Here, the basic component is the per-process IPC speedup of a configuration with prefetching relative to a configuration without prefetching. Then, workload performance is quantified with the average of these speedups. Consequently, the metric used is similar to *Aggregate Weighted Speedup (AWS)*. The difference is that the baseline is performance without prefetching and not performance in an interference-free configuration. The per-process performance degradations are quantified by reporting the largest slowdowns compared to no prefetching.

5.7 Paper C.II

Exploring the Prefetcher/Memory Controller Design Space: An Opportunistic Prefetch Scheduling Strategy
M. Grannæs, M. Jahre and L. Natvig
Submitted to: Journal of Computer Science and Technology
2010

5.7.1 Abstract

Prefetching is a well-known technique for bridging the memory gap. By predicting future memory references the prefetcher can fetch data from main memory and insert it into the cache such that overall performance is increased. Modern memory controllers reorder memory requests to exploit the 3D structure of modern DRAM interfaces. In particular, prioritizing memory requests that use open pages increases throughput significantly.

In this work, we investigate the prefetcher/memory controller design space along three dimensions: prefetching heuristic, prefetch scheduling strategy and available memory bandwidth. In particular, we evaluate 5 different prefetchers and 6 prefetch scheduling strategies. Through this extensive investigation, we observed that prior prefetch scheduling strategies often cause memory bus contention in bandwidth constrained CMPs which in turn causes performance regressions. To avoid this problem, we propose a novel prefetch scheduling heuristic called *Opportunistic Prefetch Scheduling* that selectively prioritizes prefetches to open DRAM pages such that performance regressions are minimized. Opportunistic prefetch scheduling reduces performance regressions by 6.7X and 5.2X, while improving performance by 17 % and 20 % for sequential and scheduled region prefetching, compared to the direct scheduling strategy.

5.7.2 Roles of the Authors

The roles of the authors were accurately captured by the description in Section 5.6.2.

5.8 Other Publications

In addition to the above publications, I have contributed to the following articles:

- Marius Grannæs, Magnus Jahre and Lasse Natvig, **Multi-Level Hardware Prefetching using Low Complexity Delta Correlating Prediction Tables with Partial Matching**, *5th International Conference on High Performance and Embedded Architectures and Compilers*, 2010, **Best Paper Nominee**
- Marius Grannæs, Magnus Jahre and Lasse Natvig, **Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables**, *1st JILP Data Prefetching Championship*, 2009
- Guttorm Sindre, Lasse Natvig and Magnus Jahre, **Experimental Validation of the Learning Effect for a Pedagogical Game on Computer Fundamentals**, *IEEE Transactions on Education*, 2009
- Magnus Jahre and Lasse Natvig, **Performance Effects of a Cache Miss Handling Architecture in a Multi-core Processor**, *Norwegian Informatics Conference*, 2007

Furthermore, the we have been asked to submit extensions of two of our papers to special issues of two different journals:

- Marius Grannæs, Magnus Jahre and Lasse Natvig, **Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables**, *To appear in: Journal of Instruction Level Parallelism*, 2010
- Marius Grannæs, Magnus Jahre and Lasse Natvig, **Multi-Level Hardware Prefetching using Low Complexity Delta Correlating Prediction Tables with Partial Matching**, *To appear in: Transactions on High-Performance Embedded Architectures and Compilers*, 2010

Chapter 6

Concluding Remarks

6.1 Conclusion

In this thesis, we have illustrated how managing bandwidth allocations can improve system performance. We have targeted two partially overlapping types of bandwidth: miss-bandwidth and off-chip bandwidth.

We have explored several feedback-directed techniques for managing miss bandwidth. Our starting point was Paper A.II which uses memory bus utilization measurements to control miss bandwidth allocations and improve performance. In Paper A.III, we showed that managing miss bandwidth also could improve throughput and fairness. Finally, Paper A.IV models the performance effects of miss bandwidth allocations at runtime and uses this model to allocate bandwidth. The resulting system can improve performance, throughput or fairness depending on which performance metric it is configured to optimize for. The model-based approach of Paper A.IV is powered by the *Dynamic Interference Estimation Framework (DIEF)* from Paper B.II. DIEF provides accurate estimates of the average memory latency a process would experience with exclusive access to all hardware-managed shared resources. We also provide a performance model that uses DIEF to accurately estimate what the performance of a process would be in the absence of interference.

In addition, we improve performance by using prefetching to utilize off-chip bandwidth more efficiently. Here, we choose prefetches that can be efficiently executed in the rows, columns and banks of modern DRAMs. In this way, we increase bus utilization which makes it possible to execute prefetches without severely interfering with demand reads. Furthermore, this technique reduces the performance reduction of the processes that do not benefit from prefetching.

6.2 Contributions

The main research question of this thesis is:

How, and at what cost, can performance be improved by managing resource sharing in CMP memory systems?

I approach this question through three subquestions. In this section, I review the subquestions in light of the results presented in this thesis.

6.2.1 Research Question 1

RQ1: How, and at what cost, can CMP performance be improved by managing *miss bandwidth*?

Paper A.II, A.III and A.IV shed light on this research question. These papers manage miss bandwidth allocations to reduce interference and improve performance. We provide two fundamentally different approaches to miss bandwidth management. In Paper A.II and A.III, we use an iterative approach. In these papers, miss bandwidth allocations are chosen based on measurements and evaluated in the real system. Performance feedback is used to check if the new allocation was better than the previous one. In Paper A.IV, we provide an analytical model of the performance effects of miss bandwidth allocations and use this model to make allocation decisions.

The implementation cost of the miss bandwidth allocation techniques are closely related to their complexity. For the iterative methods, we use relatively simple algorithms. Furthermore, the storage overhead is low since a few strategically placed counters are sufficient to provide the required feedback. For the model-based method, the algorithms are fairly complex, and the accurate feedback mechanisms have a moderate storage cost. Since allocation decisions are used for a long period of time, we can reduce the implementation cost with a high-latency implementation. In fact, the total storage requirement for the 4-core CMP feedback mechanisms in Paper A.IV is only 13.8 KB. The main storage cost drivers are the memory bus interference measurement techniques and the *Auxiliary Tag Directory (ATD)*.

6.2.2 Research Question 2

RQ2: How, and at what cost, can *off-chip bandwidth* management be used to increase CMP performance?

We approach this research question in both a direct and indirect manner. In the direct approach (Papers C.I and C.II), we use prefetching to provide larger bandwidth allocations to processes with predictable memory access patterns. Thus, we aim to increase the performance of these processes. By choosing prefetches to improve DRAM page locality, we increase off-chip memory bus utilization. In Paper

C.I, we increase bus utilization by piggybacking prefetches to open pages on demand reads to these pages. Since this makes prefetches cheaper in terms of latency than demand reads, it is worthwhile to issue prefetches even when prefetcher accuracy is moderately low.

In Paper C.II, we propose the opportunistic prefetch scheduling strategy. In this paper, we use a novel hardware structure called the *Page Vector Table (PVT)* to make prefetch information easily available to the memory bus scheduler. We use the PVT to issue prefetches to open pages when the scheduler closes a DRAM page. In this way, we increase the performance improvement of prefetching at the same time as we reduce the performance regressions of the processes which are not helped by prefetching. The implementation cost of these techniques is mostly due to adding a prefetcher. In addition, the PVT has a small storage cost.

Paper A.II, A.III and A.IV primarily manage miss bandwidth. Since off-chip bandwidth can be an important part of the miss bandwidth, these systems *indirectly* manage off-chip bandwidth. They differ from the Category C approaches by focusing on improving system performance rather than improving the performance of individual processes. The implementation cost of these techniques was discussed in relation to RQ1.

6.2.3 Research Question 3

RQ3: How, and at what cost, can the performance and latency effects of resource sharing be *estimated*?

We provide two methodologies for estimating the latency effects of interference. In Paper B.I, we provide an *offline* methodology for estimating interference latency. We observed that the memory bus is the main interference source for the CMPs we considered. In addition, we also found that cache capacity interference may not be a significant part of the total interference. However, alleviating cache interference can have large performance effects for some benchmarks since it can remove a large part of the total memory latency. Finally, we observed that shared cache MSHRs can have a non-negligible interference impact.

An offline approach can provide important insights, but it cannot be used for runtime resource management. Therefore, Paper B.II provides an *online* technique called the *Dynamic Interference Estimation Framework (DIEF)* which provides accurate runtime estimates of private mode memory latencies. The underlying idea is that by either measuring interference or estimating private mode latencies in each shared unit, the total interference latency can be quantified. In Paper B.II, we provide interference estimation techniques for crossbar and ring interconnects, shared caches and a DDR2 memory bus. Consequently, we explore two possible implementations of the general framework.

Paper A.IV shows how the latency measurements of DIEF can be used to provide private mode performance estimates. This is made possible by the observation that

the ability of a process to utilize the parallelism available in the memory system is very similar in the shared and private modes. Consequently, we combine shared mode *Memory Level Parallelism (MLP)* measurements with DIEF's private mode latency estimates to provide runtime estimates of private mode performance. The storage overhead of this technique was discussed in relation to RQ1.

6.3 Further Work

There are a number of possibilities for further work. Firstly, we argue in Paper A.IV that our resource allocation method can be used as long as the latency cost of allocation changes can be estimated. Consequently, we can apply the same method to conventional resource allocation mechanisms if we provide models for the latency cost of cache space and memory bus bandwidth allocations. In this way, we can leverage the global view of the CMP memory system that our models provide.

Our measurement methodologies may be used to improve the decisions made by system software (OS or virtual machine). In this context, there are a number of open questions. For instance, it is not clear which measurements system software needs and how accurate these measurements need to be. In addition, there is a trade-off between the frequency of resource allocations and the need for hardware support. The allocation policies also need to be integrated with system software.

In this thesis, we distribute bandwidth with two different mechanisms: miss bandwidth allocations and prefetching. Here, miss bandwidth allocations can be used to reduce the number of concurrent requests in the memory system. Conversely, prefetching creates additional memory requests. Thus, combining these two mechanisms to improve system performance is an interesting topic for further work. To achieve this, we need to extend our performance models to account for prefetching. In this scheme, prefetching may or may not be included in the private mode. Disabling prefetching in the private mode removes the property that private mode performance is an upper bound on the achievable shared mode performance. Conversely, enabling private mode prefetching will increase the complexity of the private mode latency estimation scheme.

So far, we have focused on workloads that are collections of single-threaded programs. However, it is likely that multi-threaded programs will become more common as developers acknowledge that they need to parallelize their programs to fully utilize a CMP for a single task. Consequently, future CMPs need to efficiently support workloads that are collections of single- and multi-threaded applications. In this case, it might be possible to adapt the degree of parallelism to sharing effects in the memory system.

6.4 Outlook

The number of processing cores on a chip is expected to increase in the future [6, 62]. It may not be worth the cost to ensure that all cores have the same shared cache latency. Therefore, future CMPs will likely have *Non-Uniform Cache Access (NUCA)* latencies [78]. Furthermore, Amdahl's law states that as the number of threads is increased, the relative impact of the serial part of the program will grow [3]. For this reason, it may be helpful to have some specialized cores for serial parts of programs and simpler cores for the parallel parts, making CMPs heterogeneous [82]. In a heterogeneous CMP, it might also be useful to have special purpose cores that can execute important program types with high performance and good energy efficiency. These possibilities indicate that future resource management systems will exist in a considerably more complicated environment than today. This both increases the need for resource allocation schemes and the complexities of implementing them.

Traditionally, programmers have been accustomed to processor manufacturers providing significant annual performance improvements while upholding the abstraction of serial execution [50]. With CMPs, programmer effort is required to fully achieve the performance potential. In the short term, we can achieve good utilization of the available hardware by concurrently executing independent programs. However, there is often a limit on how many different tasks a user needs to carry out at the same time. Consequently, the focus will likely shift towards parallel processing which enables using many of the on-chip resources for completing a single task. To achieve this, it is helpful to develop support software (e.g. libraries, runtime systems and debug tools) that facilitate rapid development of parallel programs. These systems might need hardware support to provide accurate and timely information that can be used for run-time and design-time performance optimization as well as debugging. The mechanisms and methodologies proposed in this thesis can become important primitives for such support systems.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. *SIGARCH Comput. Archit. News*, 28(2):248–259, 2000.

[2] A. Alameldeen and D. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, july-aug. 2006.

[3] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, Spring Joint Computer Conference*, pages 483–485, 1967.

[4] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM 360 Model 91: Processor Philosophy and Instruction Handling. *IBM J. Research and Development*, pages 8–24, 1967.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California at Berkeley, 2009.

[6] K. Asanovic and et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.

[7] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35:59–67, 2002.

[8] S. Belayneh and D. R. Kaeli. A Discussion on Non-Blocking/Lockup-Free Caches. *SIGARCH Comp. Arch. News*, 24(3):18–25, 1996.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[10] M. V. Biesbrouck, T. Sherwood, and B. Calder. A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 45–56, 2004.

[11] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[12] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[13] E. Bloch. The Engineering Design of the Stretch Computer. In *IRE-AIEE-ACM '59 (Eastern): Eastern Joint IRE-AIEE-ACM Computer Conference*, pages 48–58, 1959.

[14] D. Burger, J. R. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *ISCA '96: Proc. of the 23rd An. Int. Symp. on Comp. Arch.*, 1996.

[15] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth Prefetching. *SIGPLAN Notices*, 41(11), 2006.

[16] J. Casazza. Intel Core i7-800 Processor Series and the Intel Core i5-700 Processor Series Based on Intel Microarchitecture (Nehalem). White paper, Intel Corp., 2009.

[17] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 264–276, 2006.

[18] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[19] T. Chen and J. Baer. Effective Hardware-Based Data Prefetching for High-performance Processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

[20] D. Chiou, L. Rudolph, S. Devadas, and B. S. Ang. Dynamic Cache Partitioning via Columnization. Computation Structures Group Memo 430, Massachusetts Institute of Technology, 1999.

[21] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proc. of the 26th Inter. Symp. on Comp. Arch.*, pages 222–233, 1999.

[22] F. Dahlgren and P. Stenstrom. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–398, apr 1996.

[23] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., 2003.

[24] W. J. Dally and B. Towles. Route Packets, Not Wires: On-chip Inteconnection Networks. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 684–689, 2001.

[25] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Application-aware Prioritization Mechanisms for On-Chip Networks. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–291, 2009.

[26] P. Diaz and M. Cintra. Stream Chaining: Exploiting Multiple Levels of Correlation in Data Prefetching. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 81–92, 2009.

[27] H. Dybdahl. *Architectural Techniques to Improve Cache Utilization*. PhD thesis, Norwegian University of Science and Technology, 2007.

[28] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch.*, 2007.

[29] H. Dybdahl, P. Stenstrom, and L. Natvig. A Cache-Partition Aware Replacement Policy for Chip Multiprocessors. In *Proceedings of 13th International Conference of High Performance Computing (HiPC)*, pages 22–34, 2006.

[30] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches. In *MEDEA '06: Proc. of the 2006 workshop on MEmory performance*, pages 45–52, 2006.

[31] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326, 2009.

[32] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ASPLOS XV: Proc. of the 15th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[33] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A Performance Model Framework. *Computer*, 35(2):68–76, 2002.

[34] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[35] K. I. Farkas and N. P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *ISCA '94: Proc. of the 21st An. Int. Symp. on Comp. Arch.*, pages 211–222, 1994.

[36] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair Thread Scheduling for Multicore Processors. Technical report, Harvard University, 2006.

[37] A. Fedorova, M. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques*, pages 25–38, 2007.

[38] R. Gabor, S. Weiss, and A. Mendelson. Fairness and Throughput in Switch on Event Multithreading. In *MICRO 39: Proc. of the 39th Int. Symp. on Microarchitecture*, pages 149–160, 2006.

[39] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo Processor Architecture. *Intel Technology Journal*, 2006.

[40] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *SIGCOMM '96: Conf. Proc. on App., Tech., Arch., and Protocols for Comp. Com.*, pages 157–168, 1996.

[41] M. Grannæs. Bandwidth-Aware Prefetching in Chip Multiprocessors. Master's thesis, Norwegian University of Science and Technology, 2006.

[42] M. Grannæs. *Reducing Memory Latency by Improving Resource Utilization.* PhD thesis, Norwegian University of Science and Technology, 2010.

[43] M. Grannæs, M. Jahre, and L. Natvig. Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors. In *XXVI IEEE International Conference on Computer Design (ICCD)*, 2008.

[44] M. Grannæs, M. Jahre, and L. Natvig. Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables. In *Data Prefetching Championships*, 2009.

[45] M. Grannæs, M. Jahre, and L. Natvig. Multi-level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matching. In *International Conference on High-Performance Embedded Architectures and Compilers*, 2010.

[46] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive Virtual Clock: a Flexible, Efficient, and Cost-Effective QoS Scheme for Networks-on-Chip. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–279, 2009.

[47] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[48] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and More Flexible Program Analysis. In *Journal of Instruction Level Parallelism*, 2005.

[49] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. SimFlex: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture. In *SIGMETRICS Perform. Eval. Rev.*, pages 31–34, 2004.

[50] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach, Fourth Edition.* Morgan Kaufmann Publishers, 2007.

[51] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[52] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based QoS Techniques for Cache/Memory in CMP Platforms. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 479–488, 2009.

[53] M. Hill and A. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38:1612–1630, 1989.

[54] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, 1987.

[55] H. Hofstee. Power Efficient Processor Architecture and the Cell Processor. *HPCA 11: 11th Int. Symp. on High-Performance Comp. Arch.*, pages 258–262, 2005.

[56] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 13–22, 2006.

[57] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors. *Computer*, 35(2):40–49, 2002.

[58] J. Huh, D. Burger, and S. W. Keckler. Exploring the Design Space of Future CMPs. In *PACT '01: Proc. of the 2001 Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.

[59] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *MICRO 37: Proc. of the 37th An. IEEE/ACM Int. Symp. on Microarch.*, pages 343–354, 2004.

[60] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA '08: Proc. of the 35th Int. Symp. on Computer Architecture*, pages 39–50, 2008.

[61] ITRS. International Technology Roadmap for Semiconductors. `http://www.itrs.net/`, 2006.

[62] ITRS. International Technology Roadmap for Semiconductors - 2007 Edition. `http://www.itrs.net/`, 2007.

[63] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th An. Int. Conf. on Supercomputing*, pages 257–266, 2004.

[64] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07*, pages 25–36, 2007.

[65] M. Jahre. Improving the Performance of Parallel Applications in Chip Multiprocessors with Architectural Techniques. Master's thesis, Norwegian University of Science and Technology, 2007.

[66] M. Jahre and L. Natvig. Performance Effects of a Cache Miss Handling Architecture in a Multi-core Processor. In *Norwegian Informatics Conference*, 2007.

[67] M. Jahre and L. Natvig. A High Performance Adaptive Miss Handling Architecture for Chip Multiprocessors. *Transactions on High Performance Embedded Architecture and Compilation*, 4(1), 2009.

[68] M. Jahre and L. Natvig. A Light-Weight Fairness Mechanism for Chip Multiprocessor Memory Systems. In *CF '09: Proc. of the 6th ACM Conf. on Computing Frontiers*, pages 1–10, 2009.

[69] M. Jahre, M. Grannæs, and L. Natvig. A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures. In *11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 622–629, 2009.

[70] M. Jahre, M. Grannæs, and L. Natvig. DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 292–306, 2010.

[71] *DDR2 SDRAM Specification*. JEDEC Solid State Tech. Association, May 2006.

[72] L. K. John. More on Finding a Single Number to Indicate Overall Performance of a Benchmark Suite. *SIGARCH Comput. Archit. News*, 32(1):3–8, 2004.

[73] L. K. John and L. Eeckhout, editors. *Performance Evaluation and Benchmarking*. CRC Press, 2005.

[74] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[75] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. *ISCA '04: Proceedings of the 31st An. Int. Symp. on Computer Architecture*, 2004.

[76] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John. A Bandwidth-aware Memory-subsystem Resource Management using Non-invasive Resource Profilers for Large CMP Systems. In *HPCA '10: Proc. of the 16th Int. Symp. on High-Performance Comp. Arch.*, 2010.

[77] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level Storage System. *IRE Transactions on Electronic Computers*, 11(2):223–235, 1962.

[78] C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. *SIGPLAN Not.*, 37 (10):211–222, 2002.

[79] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[80] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[81] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA '81: Proc. of the 8th An. Symp. on Comp. Arch.*, pages 81–87, 1981.

[82] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, 2005.

[83] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *ISCA '05: Proc. of the 32nd Int. Symp. on Comp. Arch.*, pages 408–419, 2005.

[84] A. J. Lande. Evaluering av Chip Multiprosessor Simulatorer (in Norwegian). Master's thesis, Norwegian University of Science and Technology, Norway, June 2006.

[85] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controllers. In *MICRO '08: Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, pages 200–209, 2008.

[86] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving Memory Bank-Level Parallelism in the Presence of Prefetching. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 327–336, 2009.

[87] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 89–100, 2008.

[88] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *HPCA '08: Proc. of the 13th Int. Symp. on High-Perf. Comp. Arch.*, 2008.

[89] W. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[90] W. Lin, S. K. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching. *IEEE Transactions on Computers*, 50(11), 2001.

[91] F. Liu, X. Jiang, and Y. Solihin. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[92] G. H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 453–464, 2008.

[93] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *ISPASS*, 2001.

[94] M5 Documentation. SPEC2006 Benchmarks. `http://www.m5sim.org/wiki/index.php/SPEC2006_benchmarks`. Retrieved 11.03.2010.

[95] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Haallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[96] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *HPCA '02: Proc. of the 8th Int. Symp. on High-Performance Comp. Arch.*, page 251, 2002.

[97] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[98] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-System Timing-First Simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 108–116, 2002.

[99] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Online Prediction of Applications Cache Utility. In *Int. Conf. on Embedded Comp. Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, pages 169–177, 2007.

[100] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: A QoS Framework for CMP Architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, 2009.

[101] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *SS'07: Proceedings of 16th USENIX Security Symposium*, pages 1–18, 2007.

[102] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[103] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[104] C. Natarajan, B. Christenson, and F. Briggs. A Study of Performance Impact of Memory Controller Features in Multi-processor Server Environment. In *WMPI '04: Proc. of the 3rd Workshop on Memory Perf. Issues*, pages 80–87, 2004.

[105] K. Nesbit and J. Smith. Data Cache Prefetching Using a Global History Buffer. In *10th International Symposium on High Performance Computer Architecture, HPCA-10*, pages 96–96, 2004.

[106] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[107] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. *IEEE Micro*, 25:90–97, 2005.

[108] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An Adaptive Data Cache Prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[109] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[110] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[111] NOTUR. NOTUR Web Page. `http://www.notur.no/`.

[112] K. Olukotun and L. Hammond. The Future of Microprocessors. *Queue*, 3(7): 26–29, 2005.

[113] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. *SIGPLAN Notices*, 31(9):2–11, 1996.

[114] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *PACT '03: Proc. of the 12th Int. Conf. on Parallel Architectures and Compilation Techniques*, page 244, 2003.

[115] D. G. Perez, G. Mouchard, and O. Temam. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. In *MICRO 37: Int. Symp. on Microarchitecture*, pages 43–54, 2004.

[116] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, pages 423–432, 2006.

[117] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *ISCA '06: Int. Symp. on Comp. Arch.*, pages 167–178, 2006.

[118] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[119] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[120] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[121] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, 1995.

[122] S. L. Scott and G. S. Sohi. The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control. *IEEE Trans. Parallel Distrib. Syst.*, pages 385–398, 1990.

[123] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008*, pages 1–15, 2008.

[124] A. Settle, D. Connors, E. Gibert, and A. Gonzalez. A Dynamically Reconfigurable Cache for Multithreaded Processors. *J. Embedded Comput.*, 2(2): 221–233, 2006.

[125] J. Shao and B. Davis. A Burst Scheduling Access Reordering Mechanism. In *HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch.*, 2007.

[126] J. Shao and B. T. Davis. The Bit-Reversal SDRAM Address Mapping. In *SCOPES '05: Proc. of the 2005 Workshop on Software and Compilers for Embedded Systems*, pages 62–71, 2005.

[127] G. Sindre, L. Natvig, and M. Jahre. Experimental Validation of the Learning Effect for a Pedagogical Game on Computer Fundamentals. *IEEE Transactions on Education*, 52(1):10–18, feb. 2009.

[128] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *SIGARCH Comput. Archit. News*, 20(1):5–44, 1992.

[129] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in Computer Architecture Evaluation. *Computer*, 36:30–36, 2003.

[130] A. J. Smith. Sequential Program Prefetching in Memory Hierarchies. *Computer*, 11(12):7–21, dec. 1978.

[131] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[132] J. E. Smith. Characterizing Computer Performance with a Single Number. *Communications of the ACM*, 31(10):1202–1206, 1988.

[133] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Arch. Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[134] G. S. Sohi and M. Franklin. High-bandwidth Data Memory Systems for Superscalar Processors. In *ASPLOS-IV: Proc. of the fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, 1991.

[135] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. *SIGARCH Computer Architecture News*, 34(2):252–263, 2006.

[136] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[137] B. Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 22(4):64–71, 2002.

[138] S. Srikantaiah and M. Kandemir. SRP: Symbiotic Resource Partitioning of the Memory Hierarchy in CMPs. In *Int. Conf. on High-Performance Embedded Architectures and Compilers*, 2010.

[139] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In *ASPLOS XIII: Proc. of the 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 135–144, 2008.

[140] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors. In *MICRO-42: Proc. of the Int. Symp. on Microarchitecture*, pages 517–528, 2009.

[141] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. Technical report, University of Texas at Austin, May 2006. TR-HPS-2006-006.

[142] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A Prefetch Taxonomy. *IEEE Transactions on Computers*, 53:126–140, 2004.

[143] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical report, HP Laboratories Palo Alto, 2006.

[144] M. Thottethodi, A. Lebeck, and S. Mukherjee. Exploiting Global Knowledge to achieve Self-tuned Congestion Control for k-ary n-cube Networks. *IEEE Trans. on Parallel and Distributed Systems*, 15(3):257–272, 2004.

[145] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical report, HP Laboratories Palo Alto, 2008.

[146] J. Tuck, L. Ceze, and J. Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarchitecture*, pages 409–422, 2006.

[147] University of Tromsø. Stallo Web Page. `http://docs.notur.no/uit`.

[148] S. P. VanderWiel and D. J. Lilja. When Caches Aren't Enough: Data Prefetching Techniques. *Computer*, 30(7):23–30, 1997.

[149] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: FAirly MEasuring Multithreaded Architectures. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 305–316, 2007.

[150] D. W. Wall. Limits of Instruction-Level Parallelism. Technical report, Digital Western Research Laboratory, 1993.

[151] M. V. Wilkes. Slave Memories and Dynamic Storage Allocation. *IEEE Transactions on Electronic Computers*, 14(2):270–271, 1965.

[152] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–97, 2003.

[153] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *ISCA '09: Proc. of the 36th annual Int. Symp on Computer Architecture*, pages 174–183, 2009.

[154] Y. Xie and G. H. Loh. Scalable Shared-Cache Management by Containing Thrashing Workloads. In *Int. Conf. on High-Performance Embedded Architectures and Compilers*, 2010.

[155] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 266–277, 2005.

[156] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 339–352, 2007.

[157] X. Zhou, W. Chen, and W. Zheng. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In *PACT '09: Proc. of the 18th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 384–393, 2009.

[158] Z. Zhu and Z. Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *HPCA '05: Proc. of the 11th Int. Symp. on High-Performance Comp. Arch.*, pages 213–224, 2005.

[159] Z. Zhu, Z. Zhang, and X. Zhang. Fine-Grain Priority Scheduling on Multichannel Memory Systems. *8th Int. Symp. on High-Performance Comp. Arch.*, pages 107–116, 2002.

Appendix A

Paper A.II

A High Performance Adaptive Miss Handling Architecture for Chip Multiprocessors

Magnus Jahre and Lasse Natvig

Transactions on High-Performance Embedded Architectures and Compilers
2009

Abstract

Chip Multiprocessors (CMPs) mainly base their performance gains on exploiting thread-level parallelism. Consequently, powerful memory systems are needed to support an increasing number of concurrent threads. Conventional CMP memory systems do not account for thread interference which can result in reduced overall system performance. Therefore, conventional high bandwidth Miss Handling Architectures (MHAs) are not well suited to CMPs because they can create severe memory bus congestion. However, high miss bandwidth is desirable when sufficient bus bandwidth is available. This paper presents a novel, CMP-specific technique called the Adaptive Miss Handling Architecture (AMHA). If the memory bus is congested, AMHA improves performance by dynamically reducing the maximum allowed number of concurrent L1 cache misses of a processor core if this creates a significant speedup for the other processors. Compared to a 16-wide conventional MHA, AMHA improves performance by 12% on average for one of the workload collections used in this work.

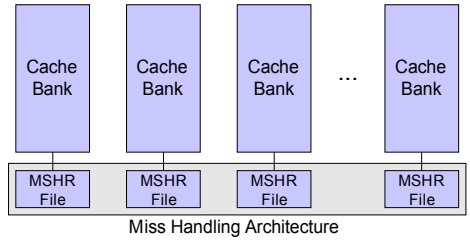


Figure A.1: Miss Handling Architecture (MHA) [29]

A.1 Introduction

Chip multiprocessors (CMPs) are now in widespread use and all major processor vendors currently sell CMPs. CMPs alleviate three important problems associated with modern superscalar microprocessors: diminishing returns from techniques that exploit instruction level parallelism (ILP), high power consumption and large design complexity. However, much of the internal structures in these multi-core processors are reused from single-core designs, and it is unclear if reusing these well-known solutions is the best way to design a CMP.

The performance gap between the processor and main memory has been growing since the early 80s [10]. Caches efficiently circumvent this problem because most programs exhibit spatial and temporal locality. However, adding more processors on one chip increases the demand for data from memory. Furthermore, latency hiding techniques will become more important and these tend to increase bandwidth demand [4].

A straightforward way of providing more bandwidth is to increase the clock frequency and width of the memory bus. Unfortunately, the number of pins on a chip is subject to economic as well as technological constraints and is expected to grow at a slow rate in the future [12]. In addition, the off-chip clock frequency is limited by the electronic characteristics of the circuit board. The effect of these trends is that off-chip bandwidth is a scarce resource that future CMPs must use efficiently. If the combined bandwidth demand exceeds the off-chip bandwidth capacity, the result is memory bus congestion which increases the average latency of all memory accesses. If the out-of-order processor core logic is not able to fully hide this latency, the result is a reduced instruction commit rate and lower performance.

It is critical for performance that the processor cores are able to continue processing at the same time as a long-latency operation like a memory or L2 cache access is in progress. Consequently, the caches should be able to service requests while misses are processed further down in the memory hierarchy. Caches with this ability are known as *non-blocking* or *lockup-free* and were first introduced by Kroft [14].

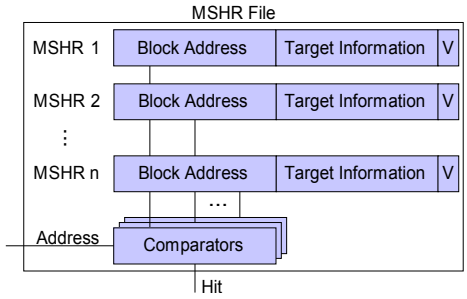


Figure A.2: A Generic MSHR File

Within the cache, the Miss Handling Architecture (MHA) is responsible for keeping track of the outstanding misses. Figure A.1 shows an MHA for a cache with multiple banks [29]. The main hardware structure within an MHA is called a *Miss Information/Status Holding Register (MSHR)*. This structure contains the information necessary to successfully return the requested data when the miss completes. If an additional request for a cache block arrives, the information regarding this new request is stored but no request is sent to the next memory hierarchy level. In other words, multiple requests for the same cache block are combined into a single memory access.

In this work, we investigate the performance impact of non-blocking caches in shared-cache CMPs and introduce a novel Miss Handling Architecture called *Adaptive MHA (AMHA)*. AMHA is based on the observation that the available miss bandwidth should be adjusted according to the utilization of the memory bus at runtime. Memory bus congestion can significantly increase the average memory access latency and result in increased lock-up time in the on-chip caches. If the processor core is not able to hide this increased latency, it directly affects its performance. Memory bus congestion reduces the performance of some programs more than others since the ability to hide the memory latency varies between programs. AMHA exploits this property by reducing the available miss bandwidth for the latency insensitive threads. Since these programs are good at hiding latency, the reduction in miss bandwidth only slightly reduces their performance. However, the memory latency experienced by the congestion sensitive programs is reduced which results in a large performance improvement. For our *Amplified Congestion Probability Workload* collection, AMHA improves the single program oriented Harmonic Mean of Speedups (HMoS) metric by 12% on average.

The paper has the following outline: First, we discuss previous work in Section A.2 before we introduce our multiprogrammed workload collections and discuss system performance metrics in Section A.3. Then, our new AMHA technique is presented in Section A.4. Section A.5 describes our experimental methodology, and Section A.6 discusses the results from our evaluation of both conventional

and adaptive MHAs. Finally, Section A.7 discusses future technology trends and possible extensions of AMHA before Section A.8 concludes the paper.

A.2 Related Work

A.2.1 Miss Handling Architecture Background

A generic Miss Status/Information Holding Register (MSHR) file is shown in Figure A.2. This structure consists of n MSHRs which contain space to store the cache block address of the miss, some target information and a valid bit. The cache can handle as many misses to *different cache block addresses* as there are MSHRs without blocking. Each MSHR has its own comparator and the MSHR file can be described as a small fully associative cache. For each miss, the information required for the cache to answer the processor's request is stored in the *Target Information* field. However, the exact *Target Information* content of an MSHR is implementation dependent. The *Valid (V)* bit is set when the MSHR is in use, and the cache must block when all valid bits are set. A blocked cache cannot service any requests.

Another MHA design option regards the number of misses to the *same cache block address* that can be handled without blocking. We refer to this aspect of the MHA implementation as *target storage*, and this determines the structure of the *Target Information* field in Figure A.2. Kroft used *implicit* target storage in the original non-blocking cache proposal [14]. Here, storage is dedicated to each processor word in a cache block. Consequently, additional misses to a given cache block can be handled as long as they go to a *different processor word*. The main advantage of this target storage scheme is its low hardware overhead.

Farkas and Jouppi [9] proposed explicitly addressed MSHRs which improves on the implicit scheme by making it possible for any miss to use any target storage location. Consequently, it is possible to handle multiple misses to *the same processor word*. We refer to the number of misses to the same cache block that can be handled without blocking as the number of targets. This improvement increases hardware cost as the offset of the requested processor word within the cache block must be stored explicitly. In this paper, we use explicitly addressed MSHRs because they provide low lock-up time for a reasonable hardware cost.

Tuck et al. [29] extended the explicitly addressed MSHR scheme to write-back caches. If the miss is a write, it is helpful to buffer the data until the miss completes which adds to the hardware overhead of the scheme. To reduce this overhead, Tuck et al. evaluated MSHRs where only a subset of the target entries has a write buffer. In addition, they extended the implicitly addressed MSHR scheme by adding a write buffer and a write mask which simplify data forwarding for reads and reduce the area cost. The target storage implementations of Tuck et al. can all be used in our AMHA scheme to provide a more fine-grained area/performance trade-off.

In this paper, we opt for the simple option of having a write buffer available to all target storage locations as this is likely to give the best performance.

In addition, Tuck et al. proposed the Hierarchical MHA [29]. This MHA provides a large amount of Memory Level Parallelism (MLP) and is primarily aimed at processors that provide very high numbers of in-flight instructions. In a CMP, providing too much MLP can create congestion in shared resources which may result in reduced performance.

Farkas and Jouppi [9] proposed the inverted MSHR organization which can support as many outstanding requests as there are destinations in the machine. Furthermore, Franklin and Sohi [26] observed that a cache line that is waiting to be filled can be used to store MSHR information. These MHAs are extremes of the area/performance trade-off and we choose to focus on less extreme MHAs. In addition, researchers have looked into which number of MSHRs gives the best performance for conventional architectures [2, 26].

A.2.2 Related Work on Bus Scheduling, Shared Caches and Feedback

Mutlu and Moscibroda [17], Nesbit et al. [18] and Rafique et al. [21] are examples of recent work that use the memory bus scheduler to improve Quality of Service (QoS). These works differ from AMHA in that they issue memory requests in a thread-fair manner while AMHA dynamically changes the bandwidth demand to utilize the shared bus efficiently. Furthermore, memory controller scheduling techniques that improve DRAM throughput are complementary to AMHA (e.g. [22, 24]).

Other researchers have focused on techniques that use shared cache partitioning to increase performance (e.g. [7, 20]). These techniques optimize for the same goal as AMHA, but are complementary since AMHA's only impact on cache partitioning is due to a reduced cache access frequency for the most frequent bus user.

Recently, a large number of researchers have focused on providing shared cache QoS. Some schemes enforce QoS primarily in hardware (e.g. [19]) while others make the OS scheduler cooperate with hardware resource monitoring and control to achieve QoS (e.g. [5]). It is difficult to compare these techniques to AMHA as improving performance is not their primary aim.

Unpredictable interactions between processors may result in performance degradation in multiprocessor systems. Feedback control schemes can be used to alleviate such bottlenecks if the reduction is due to inadequate knowledge of the state of shared structures. For instance, Scott and Sohi [23] used feedback to avoid tree saturation in multistage networks. Thottethodi et al. [28] used source throttling to avoid network saturation and controlled their policy by a feedback-based adaptive mechanism. In addition, Martin et al. [16] used feedback to adaptively choose between a directory-based and a snooping-based cache coherence protocol. AMHA

further extends the use of feedback control by using memory bus and performance measurements to guide miss bandwidth allocations.

A.3 Multiprogrammed Workload Selection and Performance Metrics

To thouroughly evaluate Miss Handling Architectures in a CMP context, we create 40 multiprogrammed workloads consisting of 4 SPEC CPU2000 benchmarks [27] as shown in Table A.1. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. We refer to these workloads as *Random Workloads (RW)*. To avoid unrealistic interference when more than a single instance of a benchmark is part of a workload, the benchmarks are fast-forwarded a different number of clock cycles if the same benchmark is run on more than one core. If there is only one instance of a benchmark in a workload, it is fast-forwarded for 1 billion clock cycles. The second time a benchmark appears in the workload, we increase the number of fast-forward clock cycles for this instance to 1.02 billion. Then, measurements are collected for 100 million clock cycles.

To investigate the performance of AMHA in the situation it is designed for, we create 40 additional workloads where this situation is more likely than in the randomly generated workload. Here, we randomly select two workloads from the 7 SPEC2000 benchmarks that has an average memory queue latency of more than 1000 processor clock cycles when running alone in the CMP. In our simulations, these benchmarks (*mcf*, *gap*, *apsi*, *facerec*, *galgel*, *mesa* and *swim*) have average queue latencies of between 1116 and 3724 clock cycles. The two remaining benchmarks are randomly chosen from the 8 benchmarks that have an average memory queue latency of between 100 and 1000 clock cycles (i.e. *wupwise*, *vortex1*, *sixtrack*, *gcc*, *art*, *gzip*, *mgrid*, *applu*). We also require that a benchmark is only used once in one workload. We refer to these workloads as *Amplified Congestion Probability Workloads (ACPW)* and they are shown in Table A.2.

Eyerman and Eeckhout [8] recently showed that the *System Throughput (STP)* and *Harmonic Mean of Speedups (HMoS)* metrics are able to represent workload performance at the system level. The STP metric is a system-oriented performance metric, and the HMoS metric is a user-oriented performance metric. Both metrics require a performance baseline where all programs receive equal access to shared resources. In this work, we give each process exactly a $\frac{1}{P}$ share of the shared cache and at least a $\frac{1}{P}$ share of the available memory bus bandwidth where P is the number of processors. To divide memory bandwidth fairly between threads, we use Rafique et al.'s Network Fair Queueing technique [21] with a starvation prevention threshold of 1. Consequently, access to the memory bus is allocated in a round-robin fashion if all processors have at least one waiting request. The formulae used

Table A.1: Randomly Generated Multiprogrammed Workloads (RW)

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	perlbmk, ammp, parser, mgrid	9	vortex1, apsi, fma3d, sixtrack	17	perlbmk, parser, applu, apsi	25	facerec, parser, applu, gap	33	gzip, galgel, lucas, equake
2	mcf, gcc, lucas, twolf	10	ammp, bzip, parser, equake	18	perlbmk, gzip, mgrid, mgrid	26	mcf, ammp, apsi, twolf	34	facerec, facerec, gcc, apsi
3	facerec, mesa, eon, eon	11	twolf, eon, applu, vpr	19	mcf, gcc, apsi, sixtrack	27	swim, ammp, sixtrack, applu	35	swim, mcf, mesa, sixtrack
4	ammp, vortex1, galgel, equake	12	swim, galgel, mgrid, crafty	20	ammp, gcc, art, mesa	28	swim, fma3d, parser, art	36	mesa, bzip, sixtrack, equake
5	gcc, apsi, galgel, crafty	13	twolf, galgel, fma3d, vpr	21	perlbmk, apsi, lucas, equake	29	twolf, gcc, apsi, vortex1	37	mcf, gcc, vortex1, gap
6	facerec, art, applu, equake	14	bzip, bzip, equake, vpr	22	mcf, crafty, vpr, vpr	30	gzip, apsi, mgrid, equake	38	facerec, mcf, parser, lucas
7	gcc, parser, applu, gap	15	swim, galgel, crafty, vpr	23	gzip, mesa, mgrid, equake	31	mgrid, eon, equake, vpr	39	twolf, mesa, eon, eon
8	swim, twolf, mesa, gap	16	mcf, mesa, mesa, wupwise	24	facerec, fma3d, applu, lucas	32	facerec, twolf, gap, wupwise	40	mcf, apsi, apsi, equake

Table A.2: Amplified Congestion Probability Workloads (ACPW)

ID	**Benchmarks**	**ID**	**Benchmarks**	**ID**	**Benchmarks**	**ID**	**Benchmarks**	**ID**	**Benchmarks**
1	mcf, apsi, applu, wupwise	9	galgel, apsi, art, gcc	17	wupwise, vortex1, apsi, gap	25	gzip, mesa, apsi, gcc	33	wupwise, apsi, art, gap
2	gzip, mcf, art, gap	10	mcf, mesa, vortex1, wupwise	18	mcf, mesa, vortex1, gcc	26	galgel, apsi, art, gcc	34	art, apsi, mgrid, gap
3	gzip, mesa, galgel, applu	11	facerec, mcf, gcc, sixtrack	19	mcf, galgel, vortex1, applu	27	facerec, vortex1, art, gap	35	swim, mesa, mgrid, wupwise
4	gzip, galgel, mesa, sixtrack	12	gzip, mcf, mesa, applu	20	mesa, applu, sixtrack, gap	28	vortex1, mcf, mesa, applu	36	facerec, mcf, art, sixtrack
5	facerec, galgel, mgrid, vortex1	13	galgel, apsi, applu, sixtrack	21	swim, mesa, art, sixtrack	29	swim, gcc, vortex1, gap	37	facerec, gzip, gcc, gap
6	gzip, mcf, mesa, art	14	swim, vortex1, apsi, art	22	swim, mcf, gcc, wupwise	30	swim, gzip, galgel, art	38	facerec, mcf, gcc, sixtrack
7	swim, apsi, sixtrack, applu	15	swim, gzip, mesa, applu	23	mesa, apsi, vortex1, sixtrack	31	swim, gzip, galgel, wupwise	39	facerec, swim, vortex1, gzip
8	facerec, swim, art, sixtrack	16	vortex1, galgel, mesa, sixtrack	24	art, galgel, mgrid, gap	32	gzip, mcf, mesa, wupwise	40	facerec, mcf, mgrid, sixtrack

Table A.3: System Performance Metrics

Metric	Formula
Harmonic Mean of Speedups (HMoS) [15]	$\frac{1}{\sum_{i=1}^{P} \frac{IPC_i^{\text{baseline}}}{IPC_i^{\text{shared}}}}$
System Throughput (STP) [25]	$\sum_{i=1}^{P} \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{baseline}}}$

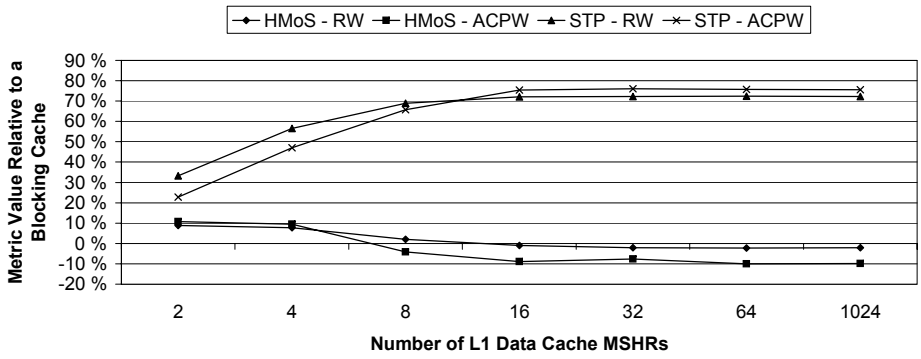


Figure A.3: Average MHA Throughput (Aggregate IPC)

to compute the HMoS and STP metrics are shown in Table A.3. The HMoS metric was originally proposed by Luo et al. [15] and the STP metric is the same as the weighted speedup metric originally proposed by Snavely and Tullsen [25].

A.4 The Adaptive Miss Handling Architecture (AMHA)

A.4.1 Motivation

Our Adaptive MHA technique is based on the observation that it is possible to drastically improve the performance of certain programs by carefully distributing miss bandwidth between threads when the memory bus is congested. This differs from earlier research on MHAs where the aim has been to provide as much miss bandwidth as possible in an area-efficient manner. Unfortunately, our results show that following this strategy can create severe congestion in the memory bus which heavily reduces the performance of some benchmarks while hardly affecting others. Figure A.3 shows the performance of a conventional MHA in a 4-core CMP plotted relative to the throughput with a blocking cache. To reduce the search space, we only modify the number of MSHRs in the L1 data cache. The 1024 MSHR

architecture is very expensive and is used to estimate the performance of a very large MHA.

Figure A.3 shows that a large conventional MHA is able to provide high throughput as measured by the STP metric. Furthermore, throughput increases with more MSHRs up to 8 MSHRs for the RW collection and up to 16 MSHRs with the ACPW collection. The reason for this difference is that there are more memory intensive benchmarks in the ACPW collection which perform better when more miss parallelism is available. Consequently, we can conclude that throughput is improved by adding more MSHRs up to 16.

The trend with the HMoS metric in Figure A.3 is very different. Here, the best values are achieved with 2 or 4 MSHRs while adding 16 or more MSHRs reduces the HMoS value below that of a blocking cache for both workload collections. The reason for this trend is that memory bus congestion does not affect all programs equally. For a memory intensive program, an increase in latency due to congestion will not create a large performance degradation. The reason is that these programs already spend a lot of their time waiting for memory. However, less memory intensive programs can hide the most of the access latency and make good progress as long as the memory latencies are reasonably low. When the memory bus is congested, the memory latencies become to large to be hidden which result in a considerable performance degradation. By carefully reallocating the available miss bandwidth, AMHA improves the performance of these latency sensitive benchmarks by reducing the available miss parallelism of the latency insensitive ones.

Figure A.3 also offers some insights into why choosing a small number of MSHRs to avoid congestion results in a considerable throughput loss. When both metrics are taken into account, the MHA with 4 MSHRs seems like the best choice. However, this results in an average throughput loss of 9% for the RW collection and 16% for the ACPW collection. In other words, simply reducing the likelyhood of congestion carries with it a significant throughput cost and an adaptive approach is needed.

A.4.2 AMHA Implementation

AMHA exploits the observation that throughput can be improved by adapting the available miss parallelism to the current memory bus utilization. Figure A.4 shows a 4-core CMP which uses AMHA. Implementing AMHA requires only small changes to the existing CMP design. First, an *Adaptive MHA Engine* is added which monitors the memory bus traffic. At regular intervals, the AMHA Engine uses run time measurements to modify the number of available MSHRs in the L1 data cache of each core. Furthermore, the MHAs of the L1 data caches are modified such that the number of available MSHRs can be changed at runtime.

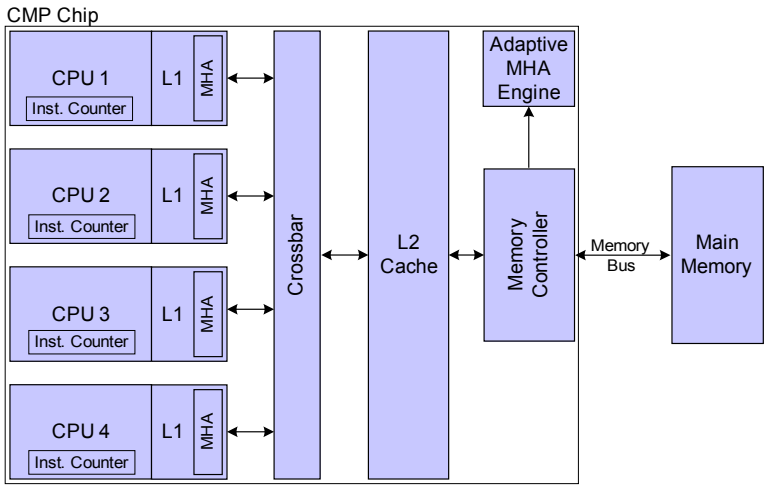


Figure A.4: General Architecture with Adaptive MHA

A.4.2.1 The AMHA Engine

Figure A.5 shows the internals of the AMHA Engine. It consists of a control unit and a set of registers called *Performance Registers*. These registers are used to measure the performance impact of an AMHA decision on all threads. In addition, the AMHA Engine stores the average bus utilization during the last sample. Here, the memory controller increments a counter each time it schedules a memory operation. The value of this counter is proportional to the actual bus utilization because each memory request occupies the data bus for a fixed number of cycles and the time between each AMHA evaluation is constant. For clarity, we will refer to this quantity as bus utilization even if AMHA uses the counter value internally. Lastly, the AMHA Engine uses a bit vector called the *blacklist* and a set of registers called *performance samples*. The blacklist is used to mark configurations that should not be tested again, and the performance samples are used to store the performance measurements for certain MHA configurations.

The performance registers store the number of committed instructions in the current and previous MHA samples. Since the number of clock cycles between each AMHA decision is constant, this quantity is proportional to IPC. These values are collected from performance counters inside each processor core when the current MHA is evaluated. By comparing these values, it is possible to estimate the performance impact of a given AMHA decision. This is necessary because it is difficult to determine the consequences of an MHA change from locally measurable variables like average queueing delays or bus utilization. The reason is that the performance with a given MHA is a result of a complex trade-off between an application's ability to hide memory latencies and its congestion sensitivity.

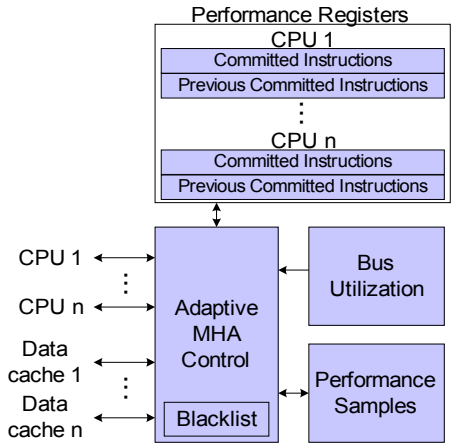


Figure A.5: Adaptive MHA Engine

A.4.2.2 Evaluating the Current MHA

Every 500000 clock cycles, the current MHA is evaluated using the information stored in the *Performance Registers*. This evaluation is carried out by the control unit and follows the pseudocode outlined in Algorithm 1. We refer to the time between each evaluation as one MHA *sample*.

To adapt to phase changes, AMHA returns all data caches to their maximum number of MSHRs at regular intervals. We refer to the time between two such resets as an *AMHA period*. After a reset, we run all data caches with their maximum number of MSHRs in one sample to gather performance statistics. If the bus utilization is lower than a configurable threshold in this sample, AMHA decides that the memory bus is not congested and turns itself off in this AMHA period. We refer to this threshold as the *Congestion Threshold*. The AMHA search procedure has a small performance impact, so we want to be reasonably certain that it is possible to find a better MHA for it to be invoked.

AMHA has now established that the memory bus is most likely congested, and it starts to search for an MHA with better performance. This search consists of two phases. In the first phase, AMHA looks for the best performing *symmetric* MHA. A symmetric MHA has the same number of MSHRs in all L1 data caches. Here, AMHA starts with the largest possible MHA and then tries all symmetric MHAs where the number of MSHRs is a power of two. At the end of each sample, AMHA stores the performance with this MHA in a *Performance Samples* register and tries the next symmetric MHA. When AMHA has tried all symmetric MHAs, the *Performance Samples* registers are analyzed and the best performing MHA is chosen. Since the performance measurements might not be representable for the whole period, we require that a smaller MHA must outperform the largest MHA by a certain percentage called the *Acceptance Threshold*. For each symmetric con-

Algorithm 1 Adaptive MHA Main Algorithm

```
1: procedure EvaluateMHA
2:     if RunCount == PERIODSIZE then
3:         Reset all MHAs to their original configuration, set phase = 1 and useAMHA = true
4:         return
5:     end if
6:     if First time in a period and no congestion then
7:         Disable AMHA in this period (useAMHA = false)
8:     end if
9:     Retrieve the current number of committed instruction from the performance counters
10:    if phase == 1 and useAMHA then                                   ▷ Search Phase 1
11:        if Symmetric MHAs remaining then
12:            Reduce the MSHRs of all L1 data caches to the nearest power of 2
13:        else
14:            Choose the best performing symmetric MHA and enter Phase 2
15:        end if
16:    else if phase == 2 and useAMHA then                              ▷ Search Phase 2
17:        if Performance improvement of last AMHA decision not acceptable and useAMHA then
18:            Roll back previous decision and add processor to the blacklist
19:        end if
20:        Find the processor with the largest MHA performance impact that is not blacklisted
21:        if Processor found then
22:            Reduce or increase the number of MSHR to the nearest power of 2
23:        else
24:            All processors are blacklisted, keep current configuration for the rest of this period
25:        end if
26:    end if
27:    Increment RunCount
28:    Move current committed instructions to previous committed instructions
29: end procedure
```

figuration, we also store the number of committed instructions for each processor. This information is used in search phase 2.

In search phase 2, AMHA attempts to improve performance by searching for an *asymmetric* MHA. Here, we adjust the MHA of one processor each time the MHA is evaluated. Since a new MHA might have been chosen in phase 1, the bus may or may not be congested. Therefore, we need to choose between increasing or decreasing the number of MSHRs in this phase. If the bus utilization is larger than the *Congestion Threshold*, AMHA assumes that the bus is congested and decreases the number of MSHRs to the nearest power of two. If not, the number of MSHRs is increased to the nearest power of two. At the end of the sample, the performance impact is computed and the MHA is either kept or rolled back. If the MHA is not accepted, the processor is blacklisted and phase 2 finishes when all processors have been added to the blacklist. To maximize the performance benefit, we start with the processor where the symmetric MHA had the largest performance impact and process them in descending order.

We use a heuristic to accept or reject an MHA change in search phase 2. If the last operation was a decrease, we sum the speedups of all processors that did not have their MSHRs reduced and compare this to the degradation experienced by the reduced processor. If the difference between the sum of speedups and the degradation is larger than the configurable *Acceptance Threshold*, the new MHA is kept. For simplicity, we use the same acceptance threshold in both search phases. If the memory bus is severely congested, reducing the number of MSHRs of a processor

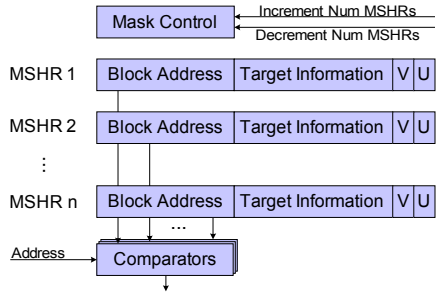


Figure A.6: The New MHA Implementation

can actually increase its performance. In this case, we set the degradation to 0. In addition, we reject any performance degradations of processors that have not had its number of MSHRs reduced as measurement errors. If the last operation increased the number of MSHRs, we sum the performance degradations of the other processors and weigh this against the performance improvement of the processor that got its number of MSHRs increased. Again, the difference must be larger than the *Acceptance Threshold* to keep the new MHA.

For each AMHA evaluation, we need to carry out P divisions in phase 1 and P divisions in phase two where P is the number of processors. The reason is that AMHA's decisions are based on relative performance improvements or degradations and not the number of committed instructions. Since there are no hard limits to when the AMHA decision needs to be ready, it can be feasible to use a single division unit for this purpose. For simplicity, we assume that the AMHA Engine analysis can be carried out within 1 clock cycle in this work. Since we need relatively large samples for the performance measurements to be accurate, it is unlikely that this assumption will influence the results. We leave investigating area-efficient AMHA Engine implementations and refining the experiments with accurate timings as further work.

A.4.2.3 MHA Reconfiguration

An MHA which includes the features needed to support AMHA is shown in Figure A.6. This MHA is changed slightly compared to the generic MHA in Figure A.2. The main difference is the addition of a *Usable (U)* bit to each MSHR. If this is set, the MSHR can be used to store miss data. By manipulating these bits, it is possible to change the number of available MSHRs at runtime. The maximum number of MSHRs is determined by the number of physical registers and decided at implementation time. As in the conventional MSHR file, the *Valid (V)* bit is set if the MSHR contains valid miss data.

The other addition needed to support AMHA is *Mask Control*. This control unit manipulates the values of the U bits subject to the commands given by the *AMHA*

Engine. For instance, if the *AMHA Engine* decides that the number of MSHRs in cache A should be reduced, cache A's *Mask Control* sets the U bits for some MSHRs to 0. In the current implementation, the number of available MSHRs is increased or decreased to the nearest power of two.

When the number of MSHRs is decreased, it is possible that some registers that contain valid miss data are taken out of use. Consequently, these registers must be searched when a response is received from the next memory hierarchy level. However, the cache should block immediately to reflect the decision of the *AMHA Engine*. This problem is solved by taking both the V and U bits into account on a cache miss and for the blocking decision. Furthermore, all registers that contain valid data (i.e. have their V bit set) are searched when a response is received.

We have chosen to restrict the adaptivity to the number of available MSHRs, but it is also possible to change the amount of target storage available. In other words, it is possible to manipulate the number of simultaneous misses to the same cache block that can be handled without blocking. This will increase the implementation complexity of AMHA considerably. Furthermore, it is only a different way to reduce the number of requests injected into the memory system. The reason is that the cache is blocked for a shorter amount of time with more targets which indirectly increases the bandwidth demand. For these reasons, AMHA keeps the amount of target storage per MSHR constant.

AMHA only requires slightly more area than a conventional MHA with the same maximum number of MSHRs as each MSHR only needs to be extended with one additional bit. Furthermore, the AMHA Engine needs a few registers and logic to compute and compare application speedups. In addition, the control functions in both the AMHA Engine and the reconfigurable MHAs require a small amount of logic.

A.5 Experimental Setup

We use the system call emulation mode of the cycle-accurate M5 simulator [3] to evaluate the conventional MHAs and AMHA. The processor architecture parameters for the simulated 4-core CMP are shown in Table A.4, and Table A.5 contains the baseline memory system parameters. We have extended M5 with an AMHA implementation, a crossbar interconnect and a detailed DDR2-800 memory bus and SDRAM model [13]. The DDR2-800 memory bus is a split transaction bus which accurately models overlapping of requests to different banks, burst mode transfer as well as activation and precharging of memory pages. When a memory page has been activated, subsequent requests are serviced at a much lower latency (page hit). We refer the reader to Cuppu et al. [6] for more details on modern memory bus interfaces. The DDR2 memory controller uses Rixner et al.'s First Ready - First Come First Served (FR-FCFS) scheduling policy [22] and reorders memory requests to achieve higher page hit rates.

Table A.4: Processor Core Parameters

Parameter	Value
Clock frequency	4 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional Units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch Predictor	Hybrid, 2048 local history registers, 2-way 2048 entry BTB

Table A.5: Memory System Parameters

Parameter	Value
Level 1 Data Cache	64 KB 8-way set associative, 64B blocks, 3 cycles latency
Level 1 Instruction Cache	64 KB 8-way set associative, 64B blocks, 16 MSHRs, 8 targets per MSHR, 1 cycle latency
Level 2 Unified Shared Cache	4 MB 8-way set associative, 64B blocks, 14 cycles latency, 16 MSHRs per bank, 8 targets per MSHR, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 8 cycles latency, 64B wide transmission channel
Memory Bus and DRAM	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [22], closed page policy

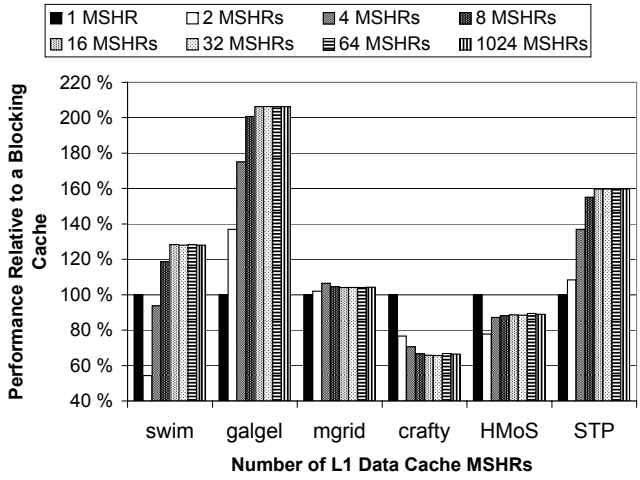
A.6 Results

A.6.1 Conventional MHA Performance in CMPs

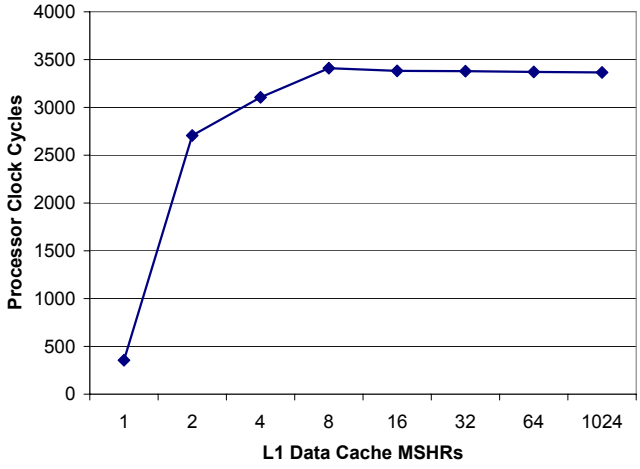
In Section A.4.1, we established that increasing the number of MSHRs improves throughput but reduces HMoS performance. However, the cause of this trend was not explained in detail. In this section, we shed some light on this issue by thoroughly analyzing the performance of the RW12 workload. This workload consists of the benchmarks *swim*, *mgrid*, *crafty* and *galgel* which are responsible for 53%, 39%, 5% and 3% of the memory bus requests with 16 MSHRs, respectively.

Figure A.7(a) shows the speedups relative to the equal allocation baseline plotted relative to the benchmark's speedup with a blocking cache configuration. In addition, the figure shows the performance trend for the system performance metrics HMoS and STP. The only benchmark that experiences a performance improvement with every increase in MHA size is *galgel*. For the other benchmarks, memory bus congestion causes more complex performance trends.

For *crafty*, performance is reduced substantially when the number of MSHRs is increased to 2. Performance is further reduced until the MHA contains 8 MSHRs before it stabilizes. Figure A.7(b) shows the average memory bus queue latency as a function of the number of MSHRs. By comparing the performance trend of



(a) Performance



(b) Average Memory Bus Queue Latencies

Figure A.7: MHA Performance with RW12

crafty with the average queue latency, we can see that for every increase in average queue latency there is a decrease in *crafty*'s performance. Since the HMoS metric is dominated by the program with the lowest performance, the HMoS metric has its highest value with the 1 MSHR MHA. However, the STP metric hides this effect and reports a throughput improvement with every increase in MHA size.

When *galgel* is provided with more MSHRs, its ability to hide the memory latencies improves enough to remove the effects of bus congestion which result in a net performance improvement. *Swim* needs a lager number of MSHRs to experience a performance improvement, but otherwise the performance trend is similar to that of *galgel*. The 2 and 4 MSHR MHAs both result in a performance reduction for *swim* because they provide to little miss parallelism to hide the long memory latencies. However, adding more MSHRs improve *swim*'s ability to hide the memory latency and result in a performance improvement. Changes in MHA size has a small performance impact on *mgrid*, and the performance difference between its best and worst MHA is only 6%.

Our study of workload RW12 has identified three properties that an adaptive MHA should be aware of. Firstly, programs with fewer memory requests are more sensitive to MHA size than memory intensive programs. Consequently, the MHA size of the memory intensive programs can be reduced to speed up the congestion sensitive programs without creating an unnecessarily large throughput degradation. Secondly, the impact of bus congestion on program performance is application dependent. Therefore, we can only rely on memory bus measurements to detect congestion while performance measurements are needed to determine the effects of an MHA change. Finally, the performance impact on an application from a change in MHA size depends on the relationship between the program's ability to hide memory latencies and the combined load the workload puts on the memory bus.

A.6.2 The Performance Impact of the Number of Targets per MSHR

Figure A.8 shows the results from varying the number of outstanding misses to the same cache block address that can be handled without blocking (i.e. the number of targets). We investigated the performance impact of varying this parameter for L1 caches with 2, 4, 8 and 16 L1 data cache MSHRs, but only report the results for the 16 MSHR case because the performance trends are very similar. The main difference is that the performance impact of adding more targets is larger with more MSHRs. If there is one target per MSHR, the cache has to block on the first miss, and this is equivalent to a blocking cache.

For both workload collections, throughput is maximized with 8 targets per MSHR. The reason is that this creates a good compromise between latency tolerance and memory bus congestion. Unfortunately, the area cost of adding 8 targets is high. Consequently, the MHA with 4 targets is probably a better choice given the small performance benefit of increasing the number of targets beyond 4. The performance

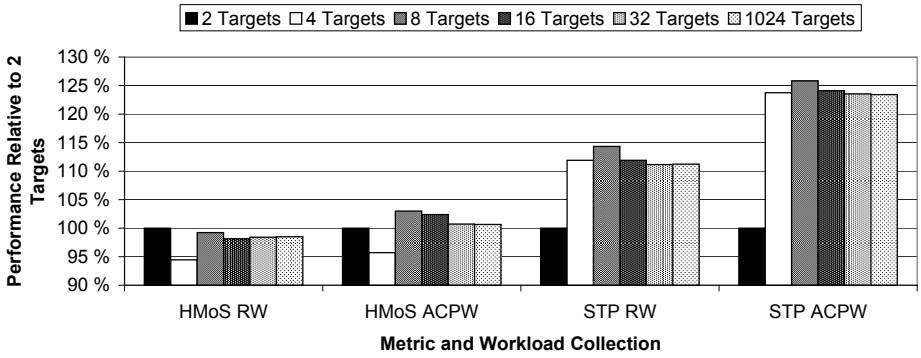


Figure A.8: Target Performance with 16 MSHRs

impact from adding more targets is larger for the ACPW collection because its workloads contain a larger number of memory intensive benchmarks by design. In other words, a greater number of benchmarks are memory intensive enough to benefit from increased miss parallelism. On the HMoS metric, adding more targets only slightly affects performance. Although the performance with 4 targets is the worst out of the examined target counts, the large increase in throughput and reasonable hardware cost makes a compelling argument for choosing this number of targets.

A.6.3 Adaptive MHA Performance

In this section, we report the results from our evaluation of the Adaptive MHA. For the experiments in this section, AMHA has a maximum of 16 MSHRs available in the L1 data cache. Therefore, the area overhead of this configuration is comparable to the conventional MHA with 16 MSHRs. AMHA only changes the number of available MSHRs in the L1 data cache for each core, and we keep the number of MSHRs in the L1 instruction caches constant at 16 for all conventional and adaptive configurations. The number of targets is 4 in all MSHRs.

AMHA aims at improving the performance of the applications that are victims of memory bus bandwidth overuse by other programs. Consequently, we expect an improvement on the HMoS metric with a reasonable reduction in system throughput. Figure A.9 shows AMHA's average performance compared to various conventional MHAs. For the RW collection, the performance impact by running AMHA is small on average, since AMHA only has a significant performance impact on 4 workloads. This is necessary for AMHA to give stable performance because reducing the number of available MSHRs can drastically reduce performance if the memory bus is not sufficiently congested. RW35 is the workload where AMHA has the largest impact with an HMoS improvement of 193% compared to a 16 MSHR MHA. If we only consider the 4 workloads where AMHA has a HMoS impact of more than 5%

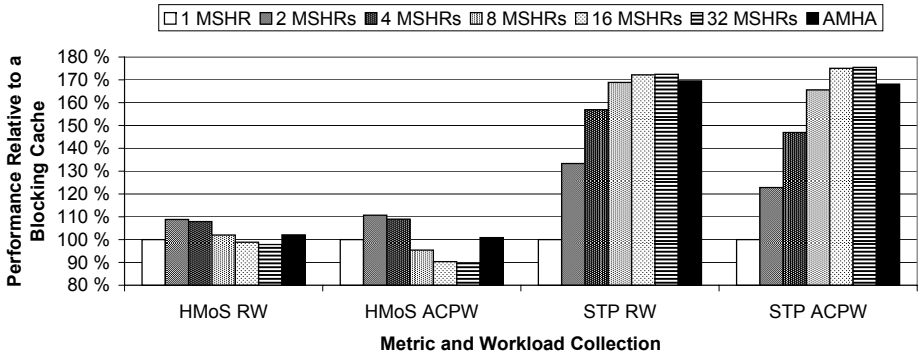


Figure A.9: AMHA Average Performance

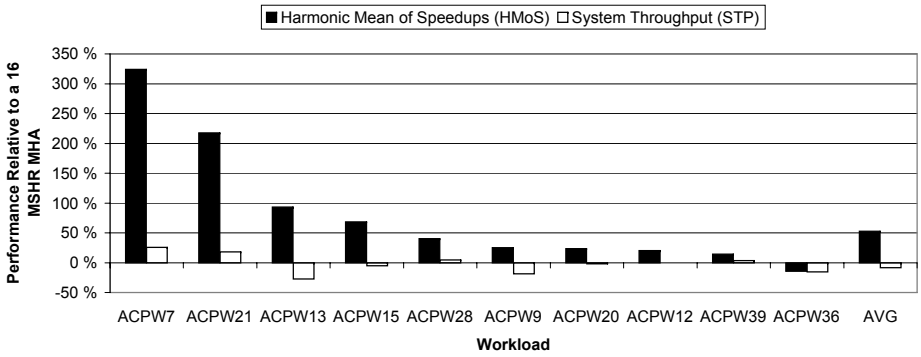
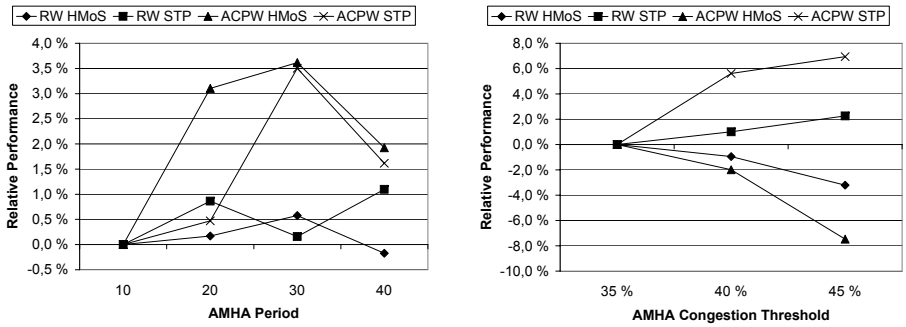


Figure A.10: AMHA Performance with High-Impact Workloads from ACPW

(both improvement and degradation), the result is an average HMoS improvement by 72% and a 3% average improvement in throughput. Consequently, we can conclude that with randomly generated workloads, AMHA has a large performance impact when it is needed and effectively turns itself off when it is not.

In the ACPW collection, the impact of AMHA is much larger since memory bus congestion is more likely for these workloads. Figure A.10 shows the performance of AMHA relative to that of a conventional 16 MSHR MHA for the workloads where AMHA has a larger HMoS impact (both improvement and degradation) of more than 10%. Again, AMHA has a large HMoS impact when it is needed and improves HMoS by 52% on average and as much as 324%. In some cases AMHA also improve STP, but the common case is a small STP degradation. Since AMHA reduces the miss bandwidth of the memory bus intensive programs, it is likely that their performance is reduced which is shown in our measurements as a throughput reduction.



(a) AMHA Period

(b) AMHA Congestion Threshold

Figure A.11: AMHA Settings

For ACPW36 (*facerec*, *mcf*, *art* and *sixtrack*), AMHA reduces both HMoS and STP. Here, bus utilization is low enough for AMHA to be turned off in most periods. However, there are two periods of bus congestion where AMHA's performance measurements indicate a large speed-up by significantly reducing *sixtrack*'s number of MSHRs. Although this is correct when the measurements are taken, AMHA keeps this configuration also after the brief period of congestion has passed. Consequently, the available miss parallelism is reduced more than necessary which results in a performance degradation on both metrics.

A.6.4 Choosing AMHA Implementation Constants

Up to now, we have used an AMHA implementation with a period of 30, a congestion threshold of 40% and an acceptance threshold of 10%. These values have been determined through extensive simulation of possible AMHA implementations. Figure A.11(a) shows the performance impact of varying the AMHA period setting. Here, the value must be chosen such that the cost of searching for a good MHA is amortized over a sufficiently long period as well as that a new search is carried our before the findings from the last search becomes obsolete. Figure A.11(b) shows the parameter space for the congestion threshold setting which adjusts the bus utilization necessary to conduct an MHA search. Here, STP is maximized with a high threshold value and HMoS is maximized with a low threshold value. Since we in this work aim at increasing HMoS while tolerating a small STP reduction, the middle value of 40% is a good choice. However, choosing 45% as the threshold is appropriate if a more throughput friendly AMHA is desired.

Finally, AMHA also needs an acceptance threshold which determines how large the difference between the performance benefit and performance cost of a sample MHA must be for the sample MHA to be used for the remainder of the AMHA period. Here, we investigated values in the 2% to 10% range and found that 10% gave the

best results. For the RW collection this parameter had nearly no impact while for the ACPW collection both HMoS and STP was maximized by choosing 10%. In general, the acceptance threshold must be large enough to filter out reduction operations that are not justified and small enough to carry out the MHA reductions when they are needed.

A.7 Discussion

AMHA works well for the CMP architecture used in this paper. However, it is important that it will also work well in future CMP architectures. Since AMHA improves performance when there is congestion in the memory bus, the performance gains are closely tied to the amount of congestion. The width of the memory bus and the clock frequency are both subject to technological constraints [12]. Consequently, it is unlikely that bus bandwidth can be improved sufficiently to match the expected increase in the number of processing cores [1]. Unless a revolutionary new memory interface solution is discovered, off-chip bandwidth is likely to become an even more constrained resource in the future [11]. Consequently, techniques like AMHA will become more important.

Currently, AMHA does not support multithreaded applications or processor cores with SMT. To support multithreaded applications, we need to treat multiple processor cores as a single entity when allocating miss bandwidth. This can be accomplished by letting the operating system provide some simplified process IDs as discussed by Zaho et al. [30] and communicate this to the Adaptive MHA Engine. Furthermore, some logic must be added to keep instructions committed in busy wait loops out of AMHA's performance measurements. Introducing SMT further complicates matters as each core now supports more than one hardware thread. Here, we need to further extend the MHA to allocate a different number of L1 MSHRs to each hardware thread. We leave the exact implementation and evaluation of such extensions as further work.

By targeting the victims of memory bus congestion and improving their performance, one might argue that AMHA is a fairness technique. However, AMHA only target unfairness in one situation, namely when the memory bus is severely congested. Furthermore, AMHA makes no guarantees of how much miss bandwidth each processor is given. Therefore, it is better to view AMHA as a simple performance optimization that can be applied when certain conditions are met.

A.8 Conclusion

When designing Miss Handling Architectures (MHAs), the aim has been to support as many outstanding misses as possible in an area efficient manner. Unfortunately, applying this strategy to a CMP will not realize its performance potential. The

reason is that allowing too much miss parallelism creates congestion in the off-chip memory bus.

The first contribution of this paper is a thorough investigation of conventional MHA performance in a CMP. The main result of this investigation was that a majority of applications need large miss parallelism. However, this must be provided in a way that avoids memory bus congestion. Our Adaptive MHA (AMHA) scheme serves this purpose and is the second contribution in this paper. AMHA increases CMP performance by dynamically adapting the allowed number of outstanding misses in the private L1 data caches to the current memory bus utilization.

Bibliography

[1] K. Asanovic and et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.

[2] S. Belayneh and D. R. Kaeli. A Discussion on Non-Blocking/Lockup-Free Caches. *SIGARCH Comp. Arch. News*, 24(3):18–25, 1996.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[4] D. Burger, J. R. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *ISCA '96: Proc. of the 23rd An. Int. Symp. on Comp. Arch.*, 1996.

[5] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[6] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proc. of the 26th Inter. Symp. on Comp. Arch.*, pages 222–233, 1999.

[7] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch.*, 2007.

[8] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[9] K. I. Farkas and N. P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *ISCA '94: Proc. of the 21st An. Int. Symp. on Comp. Arch.*, pages 211–222, 1994.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach, Fourth Edition*. Morgan Kaufmann Publishers, 2007.

[11] J. Huh, D. Burger, and S. W. Keckler. Exploring the Design Space of Future CMPs. In *PACT '01: Proc. of the 2001 Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.

[12] ITRS. International Technology Roadmap for Semiconductors. `http://www.itrs.net/`, 2006.

[13] *DDR2 SDRAM Specification*. JEDEC Solid State Tech. Association, May 2006.

[14] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA '81: Proc. of the 8th An. Symp. on Comp. Arch.*, pages 81–87, 1981.

[15] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *ISPASS*, 2001.

[16] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *HPCA '02: Proc. of the 8th Int. Symp. on High-Performance Comp. Arch.*, page 251, 2002.

[17] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[18] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[19] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[20] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, pages 423–432, 2006.

[21] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[23] S. L. Scott and G. S. Sohi. The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control. *IEEE Trans. Parallel Distrib. Syst.*, pages 385–398, 1990.

[24] J. Shao and B. Davis. A Burst Scheduling Access Reordering Mechanism. In *HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch.*, 2007.

[25] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Arch. Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[26] G. S. Sohi and M. Franklin. High-bandwidth Data Memory Systems for Superscalar Processors. In *ASPLOS-IV: Proc. of the fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, 1991.

[27] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[28] M. Thottethodi, A. Lebeck, and S. Mukherjee. Exploiting Global Knowledge to achieve Self-tuned Congestion Control for k-ary n-cube Networks. *IEEE Trans. on Parallel and Distributed Systems*, 15(3):257–272, 2004.

[29] J. Tuck, L. Ceze, and J. Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarchitecture*, pages 409–422, 2006.

[30] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 339–352, 2007.

Appendix B

Paper A.III

A Light-Weight Fairness Mechanism for Chip Multiprocessor Memory Systems

Magnus Jahre and Lasse Natvig
ACM International Conference on Computing Frontiers
2009

Abstract

Chip Multiprocessor (CMP) memory systems suffer from the effects of destructive thread interference. This interference reduces performance predictability because it depends heavily on the memory access pattern and intensity of the co-scheduled threads. In this work, we confirm that all shared units must be thread-aware in order to provide memory system fairness. However, the current proposals for fair memory systems are complex as they require an interference measurement mechanism and a fairness enforcement policy for all hardware-controlled shared units. Furthermore, they often sacrifice system throughput to reach their fairness goals which is not desirable in all systems.

In this work, we show that our novel fairness mechanism, called the Dynamic Miss Handling Architecture (DMHA), is able to reduce implementation complexity by using a single fairness enforcement policy for the complete hardware-managed shared memory system. Specifically, it controls the total miss bandwidth available to each thread by dynamically manipulating the number of Miss Status Holding Registers (MSHRs) available in each private data cache. When fairness is chosen as the metric of interest and we compare to a state-of-the-art fairness-aware memory system, DMHA improves fairness by 26% on average with the single program baseline. With a different configuration, DMHA improves throughput by 13% on average compared to a conventional memory system.

B.1 Introduction

The multi-core paradigm has become the norm for high performance processors. Commonly, these processors share part of the memory system which creates the possibility that memory requests from different processing cores can interfere with each other and increase latencies. Depending on the amount of Instruction Level Parallelism (ILP) available in the application, this can increase the processors' stall time and degrade performance. This performance reduction is unpredictable as it depends heavily on the memory access patterns and access intensity of the applications running on the other processing cores. A large, unpredictable latency variation is clearly undesirable, and an important design goal for a Chip Multiprocessor (CMP) memory system is to limit these effects by providing some form of *performance isolation*. Unfortunately, the memory systems employed in CMPs today have no means of controlling this interference, but a number of researchers have proposed techniques that alleviate this problem [3, 18, 19, 28].

A good performance isolation technique should provide both *fairness* and *Quality of Service (QoS)*. A memory system is fair if the performance reduction due to interference between threads is distributed across all processes in proportion to their priorities [14]. QoS is provided if it is possible to put a limit on the maximum slowdown a process can experience when co-scheduled with any other process [3]. Furthermore, the allowed slow-down can depend on the priority of the process. It is also common to divide fairness/QoS into *mechanisms* and *policies* [19]. Here, the policy decides the desired resource allocation and implements it with the primitives provided by the mechanism.

CMPs are commonly used in enterprise IT data centres. In this setting, it is important to have absolute control over how different threads and processes interfere with each other in order to guarantee that the resources specified in the Service Level Agreement are made available. Therefore, complex techniques that can provide QoS are needed. However, CMPs are also used in desktop computers, often running a collection of single-threaded processes. Here, a less complex solution that achieves good fairness can be more appropriate.

The main contribution of this work is a novel, light-weight mechanism called the Dynamic Miss Handling Architecture (DMHA). DMHA's key feature is that it makes it possible to change the number of Miss Status Holding Registers (MSHRs) available in the private L1 data caches at runtime. These registers determine the number of misses the cache can sustain without blocking, and a blocked cache can not receive any requests. Consequently, the processor will quickly stall as it will be unable to fetch more data, thus reducing its execution speed. DMHA uses this effect to match the execution speeds of the processors such that the slowdown due to memory system interference is equalized across threads.

DMHA divides the total miss bandwidth between all cores at the end of the private memory system. In this work, we show that DMHA is able to provide fairness values comparable to a state-of-the-art fairness-aware memory system or improve

throughput compared to a conventional memory system. We establish this result by exhaustively simulating all 256 combinations of 1, 4, 8 and 16 L1 data cache MSHRs in a 4-core CMP for 10 randomly generated multiprogrammed workloads with SPEC CPU2000 benchmarks. In other words, we measure the metric value DMHA is able to attain if provided with a good policy for this metric. A fairness policy that uses DMHA as its mechanism can improve on this result by adapting the number of MSHRs to interference patterns at runtime as well as using more fine grained MSHR allocation.

To show that DMHA can be used as the mechanism in a practical fairness implementation, we implement a simple interference measurement scheme which we call *Interference Points (IP)*. Here, we add a few registers to each shared unit and increment these with a fixed value each time we detect a certain type of interference. This measurement technique builds on previous work, and our contribution is to integrate it into a coherent whole [17, 28]. In addition, we implement a simple hardware policy that searches through different DMHA configurations at runtime to reduce interference. The combination of these techniques result in a fairness management system which we call the *Fair Adaptive Miss Handling Architecture (FAMHA)*. FAMHA is suitable for systems where strict QoS is not needed and a trade-off between fairness and throughput is desired.

The rest of this paper has the following outline. First, section B.2 gives the necessary background information on fairness techniques, MHAs and metrics. Then, section B.3 presents our DMHA mechanism before section B.4 discusses IP interference measurement and our hardware policy. Section B.5 discusses our simulation methodology before we present our results in section B.6. Finally, section B.7 discusses possible DMHA extensions before section B.8 concludes the paper and gives indications for further work.

B.2 Background

There are three types of shared resources in a typical CMP memory system: the crossbar or some other form of interconnect, one or more shared caches and a memory bus. Previous research has established that thread interference is undesirable as it can lead to unpredictable performance. Consequently, there is a need to control this interference, and researchers have investigated shared cache fairness/QoS [3, 9–11, 14, 21, 22, 28], memory bus fairness/QoS [17, 18, 20, 23] or both [2, 12, 19]. We start this section by illustrating the common points of these proposals in sections B.2.1 and B.2.2. Our novel DMHA mechanism extends the Miss Handling Architecture (MHA) of the private L1 caches to enable chip-wide allocation of miss bandwidth. Therefore, a brief introduction to MHA design is given in section B.2.3. Finally, section B.2.4 discusses the performance evaluation metrics we use in this work.

B.2.1 Shared Cache QoS and Fairness Techniques

In a shared cache, there are two resources that must be managed in order to provide fairness: *capacity* and *bandwidth*. In current CMPs, cache capacity is managed by a least recently used (LRU) policy, and cache bandwidth is distributed on a first come, first served (FCFS) basis [21]. When a cache is shared, more sophisticated techniques are needed to control the sharing of these resources. The main reason is that a thread with a higher access frequency will get a larger share of the resource with both the LRU and the FCFS policy.

The proposed cache capacity sharing techniques often use *way-partitioning* to control the cache capacity usage of each thread [11, 12, 14]. Consequently, a thread ID has to be stored in every cache block. Then, the replacement algorithm is modified to use these IDs to keep the number of blocks in a set within a quota. Normally, a single spatial partition is used as long as the running threads are in stable program phases. However, Chang and Sohi [3] observed that using multiple time sharing partitions can improve throughput compared to single partition techniques while still providing QoS. If only resource usage measurement is required, *set sampling* [28] can be used to reduce the area overhead. Here, the thread IDs are only stored for a subset of cache sets.

Nesbit et al. [21] investigated how cache bandwidth could be fairly distributed between threads. They showed that the order in which the requests are delivered to the cache must be controlled in order to provide fairness/QoS, and accomplished this by using an approach inspired by Network Fair Queueing. If there are P processors, an access latency of l cycles and all other processors have pending requests, processor A must wait $l \cdot P$ cycles between each access. Consequently, each processor is under the impression that it has a private cache that is P times slower than the shared one.

B.2.2 Memory Bus Scheduling

Memory bus scheduling is a challenging problem due to the 3D structure of DRAM consisting of rows, columns and banks. Commonly, a DRAM read transaction consists of first sending the row address, then the column address and finally receiving the data. When a row is accessed, its contents are stored in a register known as the row buffer, and a row is often referred to as a *page*. If the row has to be activated before it can be read, the access is referred to as a *row miss* or *page miss*. It is possible to carry out repeated column accesses to an open page, called *row hits* or *page hits*. This is a great advantage as the latency of a row hit is much lower than the latency of a row miss. Furthermore, a DRAM page is commonly much larger than a cache line which increases the probability of this event. DRAM accesses are pipelined, so there are no idle cycles on the memory bus if the next column command is sent while the data transfer is in progress. Furthermore, command accesses to one bank can be overlapped with data transfers from a different bank.

If data from a different row is requested, the open row must be written back into the DRAM array. This is accomplished with a *precharge* command. With a *closed page policy*, the page is written back when there are no pending requests for that row. If a row is left open until there is a request for a different row in the bank, the controller uses an *open page policy*. The situation where two consecutive requests access the same bank but different rows is known as a *row conflict*. In this case, the old row must be precharged before the row and column commands can be sent. This is very expensive in terms of latency. We refer the reader to Cuppu et al. [4] for more details regarding the DRAM interface.

Rixner et al. [24] proposed the First Ready - First Come First Served (FR-FCFS) algorithm for scheduling DRAM requests. Here, memory requests are reordered to achieve high page hit rates which results in increased memory bus utilization. This algorithm prioritizes requests according to three rules: prioritize ready commands over commands that are not ready, prioritize column commands over other commands and prioritize the oldest request over younger requests. A number of researchers have extended the FR-FCFS algorithm to handle multiple threads with different priorities [8, 17, 18, 20]. Common to these techniques is that they augment the basic FR-FCFS algorithm with additional rules so that the memory bandwidth is divided among threads in a fair manner.

B.2.3 Miss Handling Architectures

A Miss Handling Architecture (MHA) consists of one or more Miss Status/Information Holding Register (MSHR) files. The MSHR file consists of n MSHRs which contain space to store the cache block address of the miss, some target information and a valid bit. The cache can handle as many misses to *different cache block addresses* as there are MSHRs without blocking. Each MSHR has its own comparator and the MSHR file can be described as a small fully associative cache. For each miss, the information required for the cache to answer the processor's request is stored in the *Target Information* field. However, the exact *Target Information* content of an MSHR is implementation dependent. The *Valid (V)* bit is set when the MSHR is in use, and the cache must block when all valid bits are set. A blocked cache cannot service any requests.

Another MHA design option regards the number of misses to the *same cache block address* that can be handled without blocking, and we refer to this aspect of the MHA implementation as *target storage*. Kroft used *implicit* target storage in the original non-blocking cache proposal [15]. Here, storage is dedicated to each processor word in a cache block. Consequently, additional misses to a given cache block can be handled as long as they go to a *different processor word*. Farkas and Jouppi [6] proposed explicitly addressed MSHRs which improves on the implicit scheme by making it possible for any miss to use any target storage location. Consequently, it is possible to handle multiple misses to *the same processor word*. This improvement increases hardware cost as the offset of the requested processor word within the

Table B.1: CMP Performance Metrics

Metric	Formula	System-Level Meaning [5]	Reference
Aggregate Weighted Speedup (AWS)	$\sum_i^P \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{base}}}$	System Throughput (STP)	Snavely and Tullsen [25]
Harmonic Mean of Speedups (HMoS)	$\frac{P}{\sum_i^P \frac{\text{IPC}_i^{\text{base}}}{\text{IPC}_i^{\text{shared}}}}$	Inverse of Average Normalized Turnaround Time (ANTT)	Luo et al. [16]
Fairness (for one workload)	$min(\frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{base}}})/max(\frac{\text{IPC}_j^{\text{shared}}}{\text{IPC}_j^{\text{base}}})$	Assumed by system software	Gabor et al. [7]

cache block must be stored explicitly. In this paper, we use explicitly addressed MSHRs because they provide low lock-up time for a reasonable hardware cost.

B.2.4 CMP Performance Evaluation Metrics

To evaluate fairness or QoS it is necessary to identify a fair configuration which can be used as a performance baseline. We use the three metrics in table B.1 to compare the thread's performance in the shared environment with the baseline. Eyerman and Eeckhout [5] recently showed that these three metrics are sufficient to measure system throughput (STP), how fast a single program is executed (average single program turnaround time) and to what extent the effects due to sharing affect all threads equally (fairness). In table B.1, P is the total number of processors and i and j are arbitrary processor IDs. In this paper, we use the abbreviations *AWS* and *HMoS* to refer to the *Aggregate Weighted Speedup* and *Harmonic Mean of Speedups*, respectively. We use HMoS instead of Eyerman and Eeckhout's *Average Normalized Turnaround Time (ANTT)* metric because a higher value on the HMoS metric is better (ANTT is the inverse of HMoS). This makes our plots easier to read as higher is better on all metrics. When we compare the fairness of different architecture configurations, we use the arithmetic mean of the per workload fairness metric values to produce a single fairness number for each configuration.

Researchers have previously used two different fairness/QoS baselines. Firstly, it is possible to use the benchmark running alone as the baseline [2, 5, 18]. This baseline is often used when investigating memory bus fairness as any interleaving of requests might destroy page locality. For shared cache research, it is more common to compare to a static allocation where each processor is guaranteed an amount of cache space proportional to its assigned priority [3, 10]. Although this allocation gives insights into a thread's performance with a given amount of cache space, it removes all information on the thread's ability to put a larger cache capacity to good use. Since it is reasonable to assume that the choice of baseline will influence

Algorithm 2 Fairness Policy Algorithm

```
Find maximum Interference Point (IP) value
if Maximum Value > Max allowed IP then
    if Same interfering and delayed processor as last time then
        if Repeat counter > Repeat Threshold then Reduce MSHRs
        elseIncrement repeat counter
        end if
    end if
end if
```

the results, we use both baselines in this work. We will refer to them as the *Single Program Baseline (SPB)* when comparing to the benchmark running alone and the *Multiprogrammed Baseline (MPB)* when comparing to the benchmark in a configuration with equal and static shares of all resources.

B.3 The Dynamic Miss Handling Architecture

Earlier research on memory system fairness has focused on achieving fairness by dividing bandwidth or capacity between threads for a single shared unit. Our approach differs in that it allocates per thread miss bandwidth by manipulating the number of available MSHRs at runtime. Figure B.1 shows an MHA where the number of MSHRs can be dynamically reconfigured. The main difference between this MHA and a conventional MHA is the addition of a *Usable (U)* bit to each MSHR. If this is set, the MSHR can be used to store miss data. By manipulating these bits, it is possible to dynamically change the number of available MSHRs. The maximum number of MSHRs is determined by the number of physical registers and decided at implementation time. As in the conventional MSHR file, the *Valid (V)* bit is set if the MSHR contains valid miss data.

The other addition needed to support DMHA is *Mask Control*. This control unit manipulates the values of the U bits subject to the commands given by the miss bandwidth allocation policy. For instance, if the number of MSHRs in cache A should be reduced, cache A's *Mask Control* sets the U bits for some MSHRs to 0. When the number of MSHRs is decreased, it is possible that some registers that contain valid miss data are taken out of use. Consequently, these registers must be searched when a response is received from the next memory hierarchy level. However, the cache should block immediately to reflect the policy decision. This problem is solved by taking both the V and U bits into account on a cache miss and for the blocking decision. Furthermore, all registers that contain valid data (i.e. have their V bit set) are searched when a response is received.

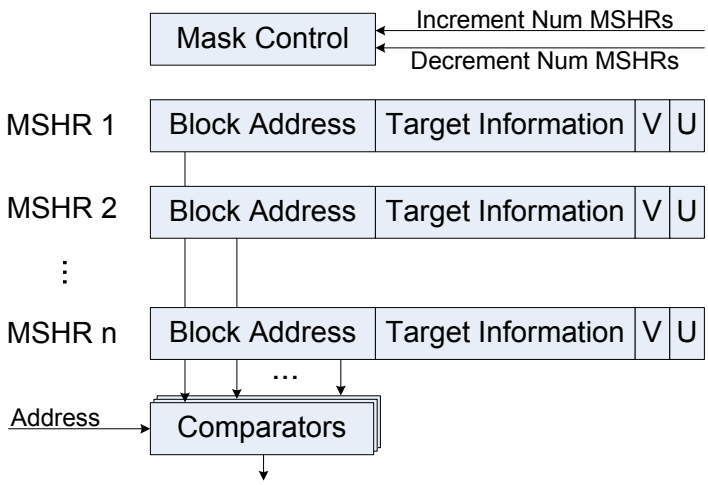


Figure B.1: Dynamic Miss Handling Architecture

B.4 The Fair Adaptive Miss Handling Architecture (FAMHA)

A practical fairness system needs to carry out three tasks: measurement, allocation and enforcement. In this section, we discuss our proposals for the measurement and allocation tasks as these are needed to use DMHA for fairness enforcement. Figure B.2 illustrates how our Interference Point (IP) measurement technique provides data to the allocation module (FAMHA Engine) which in turn controls the DMHA mechanism. Periodically, the allocation module uses the interference measurements to modify the number of MSHRs available in each private data cache. In this work, we present a simple hardware policy for the FAMHA Engine but it is also possible to implement more sophisticated software policies for flexibility.

B.4.1 Measuring Interference with Interference Points

When implementing the allocation module, it is useful that a common representation of interference is available. Consequently, we introduce the notion of *Interference Points (IPs)*. Table B.2 shows the different types of interference accounted for in our interference point measurement technique. Since each L2 cache bank in our model has one input/output channel which is connected to all L1 data and instruction cache pairs, it is contention for this channel that results in both crossbar and shared cache bandwidth interference. Furthermore, we assume that the shared cache can accept a new request every $C_{\text{cache cycle time}}$ processor clock cycles. Since the crossbar is pipelined, it can schedule a new request every $C_{\text{cache cycle time}}$ cycles and a delayed request is therefore delayed by $C_{\text{cache cycle time}}$ cycles. We add

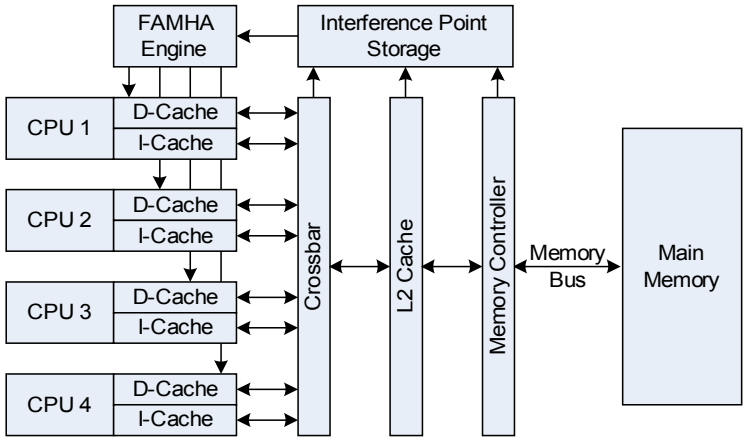


Figure B.2: Fair Adaptive MHA (FAMHA) Block Diagram

Interfering Processor

Delayed Processor	CPU 0	CPU 1	CPU 2	CPU 3
CPU 0	0	1783	21350	12930
CPU 1	1928	0	18795	14706
CPU 2	2750	1429	0	12254
CPU 3	5847	4273	18604	0

Figure B.3: Interference Point Storage

$C_{\text{cache cycle time}}$ for each processor which has one or more delayed requests. The reason is that misses that are clustered together usually have a smaller performance impact than solitary misses.

Interference due to contention for cache capacity is an important source of interference in CMP memory systems. However, it is difficult to estimate the extra delay resulting from one processor exceeding its cache space quota. Since our hardware fairness policy is simple, we choose the low complexity option of using the number of blocks the processor is using beyond its equal allocation baseline as the interference point value. This is easy to measure as it simply consists of adding a register for each processor which is incremented when the processor brings a block into the cache and decremented on a replacement. Zaho et al. [28] showed that such measurements could be implemented with a small area overhead by using set sampling. In this work, we assume that we know which processor brought each block into the cache which results in an area overhead comparable to that of a way-partitioned cache fairness technique. We also avoid the problem of estimating the impact of this overuse on the other processors by assuming that it affects all processors by an

Table B.2: Interference Point Formulae

Shared Resource	**Requirement**	**IP Value**
Crossbar Bandwidth	At least one request is delayed	$C_{\text{cache cycle time}}$
Shared Cache Capacity	The processor uses more than its static share	$max(\text{Occupied Blocks} - \frac{\text{Total Blocks}}{P}, 0)$
Memory Bus Bandwidth [17]	At least one ready request is delayed	C_{bus}
	Row conflict	$\frac{(C_{\text{row}}+C_{\text{col}}+C_{\text{precharge}})}{\text{BankParallelism}(i)}$

equal amount. If more accurate measurements are needed, it is possible to include the interference and sharing measurement techniques proposed by Zaho et al. [28].

Mutlu and Moscibroda [17] recently proposed a low overhead scheme for measuring memory bus interference. We use a simplified version of their technique in this work. Firstly, we account for interference due to accesses being serialized on the memory bus. In this case, we add an IP quantity that corresponds to the number of processor cycles used to transfer one last-level cache block over the memory bus (C_{bus}). Secondly, processor A might have a request for a bank in which processor B already has an activated page. This situation is known as a row conflict. Consequently, it is necessary to precharge the bank before sending the row address and column address of processor A's request. Here, we approximate the actual delay by adding $C_{\text{precharge}} + C_{\text{row}} + C_{\text{col}}$ cycles. This is only an approximation since the actual additional delay may vary depending on the number of cycles the bank has been in the read or write states [13]. Furthermore, the cost of this delay can be amortized over requests to other banks. Therefore, we use Mutlu and Moscibroda's bank parallelism estimator to reduce the impact of this factor depending on the number of requests processor A has waiting for other banks.

We are now left with a collection of cycle-based and block-based interference measurements. Consequently, there is a need to combine these in a meaningful way. Generally, the total interference points follow the formula $\text{IP}_{\text{total}} = \alpha \cdot \text{IP}_{\text{cycles}} + \beta \cdot \text{IP}_{\text{blocks}}$. Consequently, the constants α and β should be chosen to reflect the relative importance of the cycle-based and block-based measurements. Since interference measurement is not the main focus of this work, we use $\alpha = 1$ and $\beta = 1$. This was sufficient for our simple allocation technique, but more sophisticated policies might need better control of the relative impact of block- and cycle-based metrics.

The interference point storage structure is shown in Figure B.3. Each shared unit has one such structure, and each entry is incremented when interference is detected. At regular intervals, the information is read by the FAMHA Engine and the counters are reset. The values on the diagonal are always zero, and it is not necessary to allocate storage for these values. The IP structure is similar to Zaho et al.'s interference tables [28]. The main difference is that Zaho et al. only record cache

Table B.3: Processor Core Parameters

Parameter	Value
Clock frequency	4 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 2-way 2048 entry BTB

capacity interference. Our interference tables stores an interference point value which makes it possible to compare different forms of interference.

B.4.2 A Simple Fairness Policy

To verify that our DMHA mechanism can be used in a practical system, we created a simple hardware policy that uses our interference points measurement technique and the DMHA mechanism to improve CMP memory system fairness. Every 500000 clock cycles, the FAMHA Engine gathers the interference points from all shared units. Then, it follows a greedy algorithm (Algorithm 2) to determine which processor should have its number of MSHRs reduced if any. At regular intervals, all processors are restored to their maximum number of MSHRs to adapt to application phase changes.

To control the aggressiveness of the adaptive policy, we add two additional configuration parameters. First, we require that the largest interference point value must be larger than a threshold for an MSHR reduction to be considered. This parameter is necessary to avoid reducing the MSHRs when there is little interference. Secondly, we require that the greedy algorithm returns the same interfering processor and delayed processor a configurable number of times before the MSHR reduction is carried out. A high value on this threshold both guards against making wrong decisions and reduces the speed with which the number of MSHRs is reduced.

B.5 Evaluation Methodology

We use the system call emulation mode of the cycle-accurate M5 simulator [1] for our experiments. The processor architecture parameters for the simulated 4-core CMP are shown in table B.3, and table B.4 contains the baseline memory system parameters. We have extended M5 with a FAMHA implementation, a crossbar interconnect and a detailed DDR2-800 memory bus and DRAM model [13]. The

Table B.4: Memory System Parameters

Parameter	**Value**
Level 1 Data Cache	64 KB 8-way set associative, 64B blocks, 16 MSHRs, 3 cycles latency
Level 1 Instruction Cache	64 KB 8-way set associative, 64B blocks, 16 MSHRs, 1 cycle latency
Level 2 Unified Shared Cache	8 MB 16-way set associative, 64B blocks, 18 cycles latency, 16 MSHRs per bank, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 8 cycles latency, 64B wide
Main memory	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [24], Closed page policy

Table B.5: List of Acronyms

AWS	Aggregate Weighted Speedup	*HMoS*	Harmonic Mean of Speedups	*MTP*	Multiple Time Sharing Partitions [3]
CB	Crossbar	*IP*	Interference Point	*NFQ*	Network Fair Queueing [20]
DMHA	Dynamic MHA	*MHA*	Miss Handling Architecture	*QoS*	Quality of Service
FAMHA	Fair Adaptive MHA	*MPB*	Multiprogrammed Baseline	*SPB*	Single Program Baseline
FR-FCFS	First Ready - First Come First Served	*MSHR*	Miss Status/Information Holding Register		

shared cache is pipelined and can accept a new request every 2 clock cycles. This value is based on the cycle time given by the CACTI cache timing analysis tool [27].

We have implemented a state-of-the-art fairness-aware memory system to evaluate our FAMHA technique. To manage cache capacity, we use Chang and Sohi's Multiple Time-Sharing Partitions (MTP) [3] which has been shown to outperform cache capacity sharing that rely on a single spatial partition. To gather the miss rate curves for each processor, we employ an auxiliary tag directory for each processor core as suggested by Chang and Sohi. However, we have not implemented their Cooperative Caching throughput optimization because it can not be applied to shared caches where all banks have a uniform latency.

Furthermore, we use two variants of Rafique et al.'s state-of-the-art, thread-aware memory bus scheduling scheme based on Network Fair Queueing (NFQ) [23]. NFQ-1 allows no access reordering while NFQ-3 allows at most three requests to pass the request with the lowest virtual start time. Our fair crossbar provides fairness with Start Time Fair Queueing [8]. It also provides fair cache bandwidth allocation because the crossbar serializes requests to the L2 banks in our model. The fair crossbar of Nesbit et al. [21] is different from ours since it allocates cache bandwidth with virtual deadline first scheduling.

Table B.6: Randomly Generated Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	perlbmk, ammp, parser, mgrid	9	vortex1, apsi, fma3d, sixtrack	17	perlbmk, parser, applu, apsi	25	facerec, parser, applu, gap	33	gzip, galgel, lucas, equake
2	mcf, gcc, lucas, twolf	10	ammp, bzip, parser, equake	18	perlbmk, gzip, mgrid, mgrid	26	mcf, ammp, apsi, twolf	34	facerec, facerec, gcc, apsi
3	facerec, mesa, eon, eon	11	twolf, eon, applu, vpr	19	mcf, gcc, apsi, sixtrack	27	swim, ammp, sixtrack, applu	35	swim, mcf, mesa, sixtrack
4	ammp, vortex1, galgel, equake	12	swim, galgel, mgrid, crafty	20	ammp, gcc, art, mesa	28	swim, fma3d, parser, art	36	mesa, bzip, sixtrack, equake
5	gcc, apsi, galgel, crafty	13	twolf, galgel, fma3d, vpr	21	perlbmk, apsi, lucas, equake	29	twolf, gcc, apsi, vortex1	37	mcf, gcc, vortex1, gap
6	facerec, art, applu, equake	14	bzip, bzip, equake, vpr	22	mcf, crafty, vpr, vpr	30	gzip, apsi, mgrid, equake	38	facerec, mcf, parser, lucas
7	gcc, parser, applu, gap	15	swim, galgel, crafty, vpr	23	gzip, mesa, mgrid, equake	31	mgrid, eon, equake, vpr	39	twolf, mesa, eon, eon
8	swim, twolf, mesa, gap	16	mcf, mesa, mesa, wupwise	24	facerec, fma3d, applu, lucas	32	facerec, twolf, gap, wupwise	40	mcf, apsi, apsi, equake

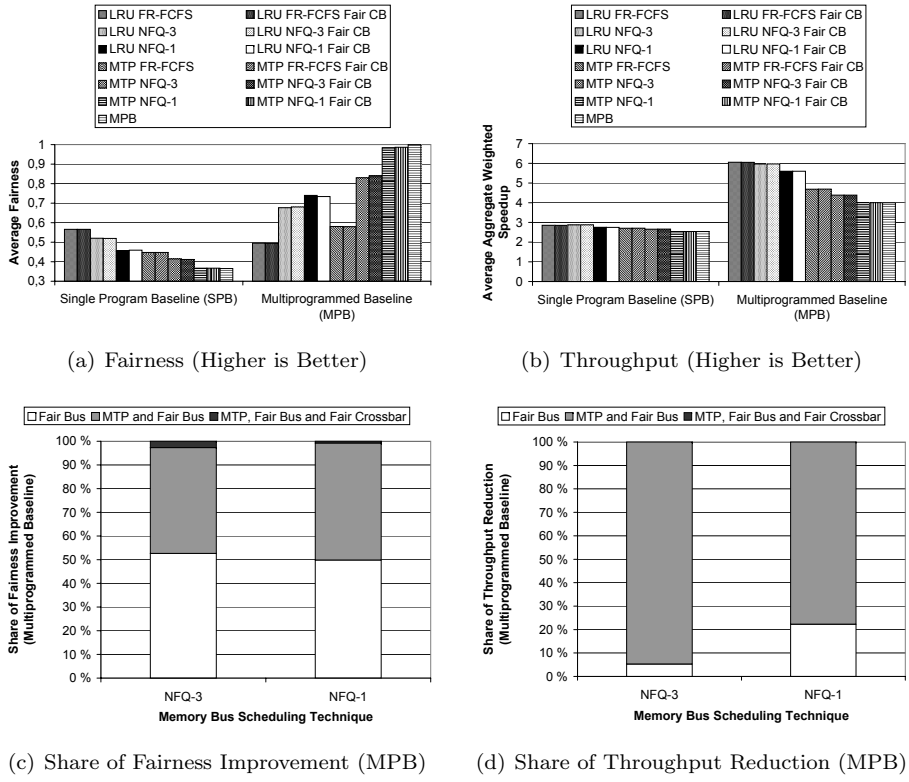
We use the SPEC CPU2000 benchmark suite [26] to create 40 multiprogrammed workloads consisting of 4 SPEC benchmarks each as shown in table B.6. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. To avoid unrealistic interference when more than a single instance of a benchmark is part of a workload, the benchmarks are fast-forwarded a different number of clock cycles if the same benchmark is run on more than one core. If there is only one instance of a benchmark in a workload, it is fast-forwarded for 1 billion clock cycles. Each time the benchmark is repeated, we increase the number of fast-forward clock cycles by 20 million. Then, measurements are collected for 200 million clock cycles.

B.6 Results

In this section, we evaluate the fairness and throughput impact of conventional fairness schemes and our new FAMHA technique. First, we quantify the relative impact of unfairness in the different shared units in section B.6.1. In section B.6.2, we quantify the potential of the DMHA mechanism by simulating a large number of static asymmetric MHAs. Here, we show that DMHA can provide similar (MPB) or better (SPB) fairness as well as better performance and throughput than a CMP with a state-of-the-art fairness enabled memory system. Finally, we show that using DMHA with a simple measurement and allocation technique substantially improves fairness compared to a conventional memory system in section B.6.3.

B.6.1 Fairness Impact of Shared Hardware-Managed Units

When designing a fair memory system, it is helpful to identify the relative fairness impact of interference in the different shared units. Figure B.4 provides some insights into this issue. Here, we report the fairness and throughput of the selected fairness techniques and quantify their relative impact on fairness and throughput. As expected, employing stricter fairness techniques improves fairness for the multiprogrammed baseline (MPB) in Figure B.4(a). However, the stricter fairness enforcement techniques actually yield lower fairness with the single program baseline (SPB). With SPB, slowdowns should be proportional to the performance of the application when running alone which is difficult to achieve due to the applications' varying sensitivity to resource allocations. A resource allocation sensitive application might experience a severe slowdown with a static share while the performance of an allocation insensitive thread would hardly change. If these threads are run together, there is a large variation in their slowdowns relative to the baseline which is interpreted as unfairness. Techniques that rely on resource partitioning tend to make these problems worse, because they limit the resources available to resource sensitive applications.



(a) Fairness (Higher is Better)

(b) Throughput (Higher is Better)

(c) Share of Fairness Improvement (MPB)

(d) Share of Throughput Reduction (MPB)

Figure B.4: Performance Impact of a Fair Memory Bus, Fair Crossbar and Fair Cache

As expected, Figure B.4(b) shows that stricter enforcement of fairness reduces system throughput. The maximum AWS value is 4 for SPB (i.e. equal to the number of processors). The reason is that SPB balances the shared mode performance against the performance with exclusive access to all shared resources. For MPB, the benchmark can seize more resources in the shared mode than is available in the baseline. Consequently, it is difficult to set a concrete bound on the AWS value for this baseline.

Figure B.4(c) shows the relative impact of each fairness technique with MTB. Here, the memory bus controller and cache capacity sharing technique each account for about 50% of the total fairness improvement with both the NFQ-1 and NFQ-3 controllers. However, the NFQ-3 controller is able to carry out this fairness increase with a very low impact on system throughput as shown in Figure B.4(d). Consequently, the fair cache sharing technique is responsible for 95% of the throughput loss due to fairness with MPB. With the NFQ-1 scheduler, the cache is responsible for 78% of the throughput loss.

The cache is responsible for most of the throughput loss because of the focus on Quality of Service (QoS). In this case, performance should never drop below a given baseline. Chang and Sohi [3] define that QoS is achieved if the value on their QoS metric ($\sum_i^P min(0, \frac{IPC_i^{\text{shared}}}{IPC_i^{\text{MPB}}} - 1)$) is larger than -0.05 for all workloads. In our experiments, only the configuration with the MTP cache, NFQ-3 bus scheduling and the fair crossbar achieves this goal. However, this configuration also reduces system throughput by 7% (SPB) and 28% (MPB) on average.

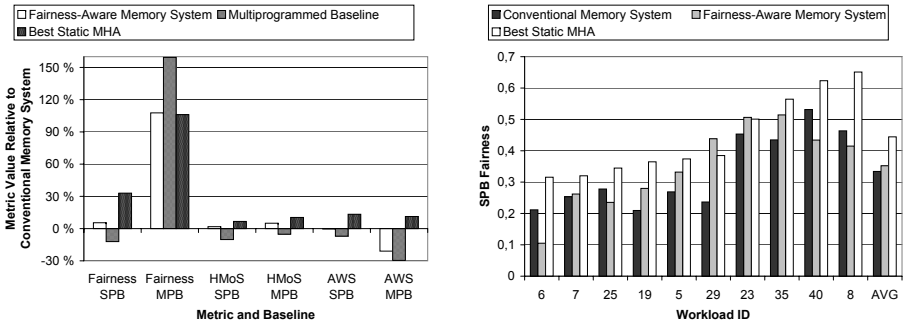
The configuration with the NFQ-1 bus does not provide QoS because MTP assumes that a thread's performance is inversely proportional to its miss rate. In workload 16, this assumption does not hold because the total number of misses is increased by MTP's throughput optimization. This creates severe memory bus congestion, and results in a slowdown for 3 out of 4 benchmarks. Note that the fairness metric also takes into account that the performance impact from sharing should affect all threads equally which results in the NFQ-3 controller having considerably poorer fairness than NFQ-1 in Figure B.4(a).

Our results suggest that the fairness impact of introducing a fair crossbar is very small. This differs from the results of Nesbit et al. [21] who reported a HMoS increase of 10% on average by implementing fair cache bandwidth sharing. We believe that this difference is due to different cache modeling assumptions. In our cache, all accesses take the same number of clock cycles. The cache is also heavily pipelined, and we do not account for any resource dependencies.

B.6.2 Static Asymmetric MHA Fairness

A good fairness mechanism should be able to achieve good fairness, throughput and single program turn around time. This makes it possible to create a policy that optimizes for the metric of interest which may vary from system to system. In this section, we show that our DMHA mechanism meets this requirement. To evaluate DMHA, we simulate all possible asymmetric L1 data cache MHAs with 1, 4, 8 and 16 MSHRs (i.e. 256 possible MHAs in a 4-core CMP). We retrieve the best value for a given metric and workload and refer to this as the *offline-best-static MHA*. Note that the configuration that yields the best performance with one metric does not necessarily yield the best performance on a different metric. This is appropriate as the aim of the experiment is to show that an asymmetric MHA can provide good performance on a given metric when provided with an appropriate policy for this metric. An adaptive policy might also outperform offline-best-static by dynamically changing the asymmetric MHA to adapt to application phase changes.

Simulating many combinations of static MHAs quickly become computationally infeasible. Consequently, we have selected 10 of our 40 randomly generated workloads for this experiment. Specifically, workloads 5, 6, 7, 8, 19, 23, 25, 29, 35 and 40 are used. These workloads all have a fairness value of 0.5 or less for the conventional memory system with both baselines.



(a) Average Performance and Fairness

(b) Fairness with the Single Program Baseline (SPB)

Figure B.5: Offline Best Static MHA Performance (Workload Subset)

Figure B.5 shows the performance of the offline-best-static MHA relative to a conventional memory system with no cache partitioning control, a FR-FCFS memory bus scheduler and conventional crossbar. In addition, the values for the best performing fairness-aware memory system (MTP cache partitioning, NFQ-3 bus scheduler and a conventional crossbar) and the multiprogrammed baseline are shown. Figure B.5(a) shows the average values of the different metrics for the subset of the randomly generated workloads relative to the conventional memory system. With SPB, offline-best-static MHA improves fairness by 26% compared to the best performing fairness-aware memory system, and it improves AWS by 13% compared to the conventional memory system. In addition, it performs better than both the conventional and the fairness-aware memory system on all metrics and baselines except fairness with MPB. In this case, the offline-best-static MHA is not able to mirror the per-core performance with the static resource sharing. This result is mainly due to workload 6 where the L1 data caches of all processors that contribute to interference have been reduced to a blocking configuration. Consequently, it is not possible to reduce performance of these benchmarks enough to match the performance with a statically shared memory system. However, offline-best-static MHA achieves fairness values comparable to the fairness-enabled memory system.

Figure B.5(b) shows the fairness results for the selected workloads with the SPB baseline. Here, the offline-best-static MHA outperforms both the conventional and fairness-enabled memory systems for 8 of 10 workloads. This indicates that a good DMHA policy should be able to approach the fairness of today's state-of-the-art fairness systems. In workload 23, *mgrid* (4 MSHRs in offline-best-static) is allowed to use enough shared cache space to create a slowdown for *gzip* (1 MSHR), *mesa* (4 MSHRs) and *equake* (1 MSHR). However, reducing *mesa's* number of MSHRs beyond 4 slows it down sufficiently to reduce overall fairness. The same problem is responsible for the less than ideal performance in workload 29. Here, cache interference between *apsi* (4 MSHRs) and *gcc* (16 MSHRs) reduces fairness

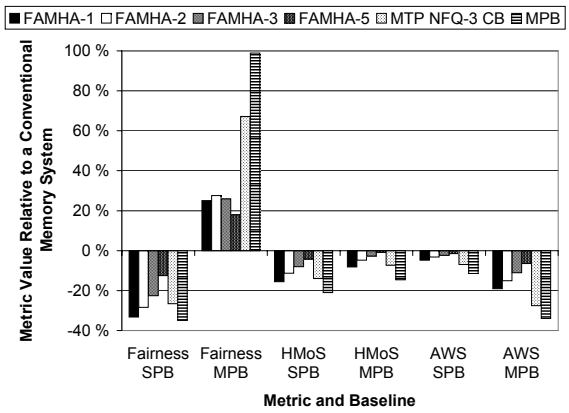


Figure B.6: FAMHA Results

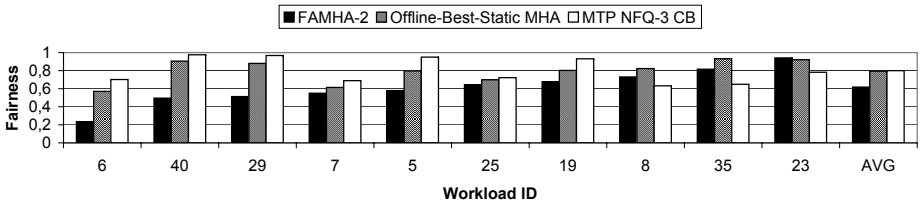


Figure B.7: FAMHA-2 Fairness with the Multiprogrammed Baseline (Workload Subset)

regardless of what number of MSHRs are assigned to them. Consequently, none of the asymmetric MHAs used by offline-best-static achieves good fairness. However, it is possible that a more thorough search would uncover an asymmetric MHA with better fairness than the ones evaluated here.

B.6.3 Fair Adaptive MHA (FAMHA) Results

In the previous section, we established that our DMHA mechanism can achieve good results when an appropriate policy is provided. Here, we report the results of the full FAMHA system which uses the IP measurement technique and the greedy allocation policy. Figure B.6 shows the average values of all metrics with our best performing FAMHA policy, the best fairness-enabled memory system (MTP, NFQ-3 and a conventional crossbar) and the multiprogrammed baseline. The FAMHA configurations evaluated here resets the MSHRs after 40 events (20 million clock cycles) and allows interference point values up to 5000. We investigated the impact of varying these parameters and found that it was small as long as FAMHA is

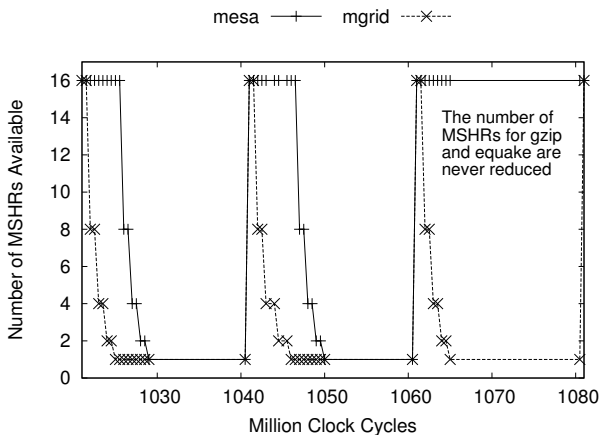


Figure B.8: Workload 23 FAMHA-2 Behaviour

given enough time to find a good solution. Furthermore, the allowed number of interference points should not be too large. The difference between the FAMHA configurations shown in Figure B.6 is the number of times FAMHA must repeat a decision before reducing the number of MSHRs.

The aim of FAMHA is to achieve good results on all metrics. FAMHA-2 achieves the best fairness with a 28% (MPB) improvement over the conventional memory system. However, this results in a reduction in single thread turn around time (HMoS) of 11% (SPB) and 5% (MPB) as well as a throughput (AWS) reduction of 3% (SPB) and 15% (MPB). FAMHA-5 is the best performing configuration when all metrics are taken into account. Here, fairness is improved by 18% (MPB) with a reduction in single thread turn around time of 4%(SPB) and 1%(MPB) and a throughput reduction of 1%(SPB) and 6%(MPB). As a comparison, the fairness-enabled memory system improves fairness by 67% (MPB). However, the cost is significant: single thread turn around time is reduced by 14%(SPB) and 7%(MPB) and throughput is reduced by 7%(SPB) and 28%(MPB).

To better understand how FAMHA impacts the fairness of a single workload, we show FAMHA-2's MPB fairness for the subset of workloads used to create the offline-best-static MHA in Figure B.7. FAMHA performs as well as can be expected and reduces fairness by 22% on average compared to the offline-best-static MHA. For workloads 8, 23 and 35, FAMHA outperforms the fairness enabled memory system. FAMHA achieves poor fairness on workload 6 which consists of the benchmarks *facerec*, *art*, *applu* and *equake*. Since offline-best-static MHA performs well, this is due to an inadequate policy. The offline-best-static MHA uses 16 MSHRs for *equake* and a blocking cache for the other benchmarks. FAMHA eventually reaches the same solution, but it is too late to achieve good fairness values. Consequently, a more aggressive version of our algorithm would be appropriate for this workload. However, this would degrade performance on other workloads.

Figure B.8 shows FAMHA-2's behaviour with workload 23 in three allocation periods and illustrates how it can outperform offline-best-static MHA. Here, the best static solution for fairness gives all applications a blocking cache. However, *mesa* and *mgrid* are the major contributors to interference. FAMHA-2 always reduces *mgrid* directly to a blocking cache configuration in all periods while *mesa* is reduced to the blocking configuration in 4 out of 10 periods. Consequently, FAMHA-2 reduces the impact of short periods of interference and the result is that it outperforms the best static MHA.

B.7 Discussion

Currently, FAMHA does not support multithreaded applications or processor cores with SMT. To support multithreaded applications, we need to treat multiple processors as a single entity when allocating miss bandwidth. This can be accomplished by letting the operating system provide some simplified process IDs as discussed by Zaho et al. [28] to the measurement scheme and resource allocation process. Introducing SMT further complicates matters as each core now supports more than one hardware thread. Here, we need to further extend the dynamic MHA to allocate a different number of L1 MSHRs to each hardware thread. We leave the exact implementation and evaluation of such extensions as further work.

B.8 Conclusion and Further Work

In this work, we introduced a novel, light-weight fairness mechanism called the *Dynamic Miss Handling Architecture (DMHA)*. By simulating a large number of static asymmetric MHAs, we showed that the DMHA mechanism can be used to provide good fairness, throughput or single program turnaround time. This result assumes that an appropriate policy is provided, and we introduced a simple policy which improves fairness considerably compared to a conventional memory system. Our policy relies on an interference measurement technique that makes it possible to coherently compare different forms of interference. Together, these techniques form a radically different approach to fairness which we call the *Fair Adaptive Miss Handling Architecture (FAMHA)*.

There are many possibilities for further work. One direction is to investigate different policies to establish the practical limits on achieving fairness with a DMHA. To achieve this, it is probably necessary to verify that our interference measurement mechanism accurately captures all forms of interference and weights them appropriately. In particular, we plan to investigate the weighting of the cycle based and block based measurements further. Furthermore, it is possible to integrate the DMHA mechanism with other light-weight mechanisms to improve fairness beyond what DMHA can achieve on its own. Finally, support for priorities, SMT processing cores and multithreaded applications should be added.

B.9 Acknowledgements

We extend our gratitude to Marius Grannaes and the anonymous reviewers for valuable comments on earlier versions of this paper. Furthermore, this work would not have been possible without the computing resources granted us by the Norwegian Metacenter for Computational Science (NOTUR).

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[3] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[4] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proc. of the 26th Inter. Symp. on Comp. Arch.*, pages 222–233, 1999.

[5] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[6] K. I. Farkas and N. P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *ISCA '94: Proc. of the 21st An. Int. Symp. on Comp. Arch.*, pages 211–222, 1994.

[7] R. Gabor, S. Weiss, and A. Mendelson. Fairness and Throughput in Switch on Event Multithreading. In *MICRO 39: Proc. of the 39th Int. Symp. on Microarchitecture*, pages 149–160, 2006.

[8] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *SIGCOMM '96: Conf. Proc. on App., Tech., Arch., and Protocols for Comp. Com.*, pages 157–168, 1996.

[9] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[10] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 13–22, 2006.

[11] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th An. Int. Conf. on Supercomputing*, pages 257–266, 2004.

[12] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07*, pages 25–36, 2007.

[13] *DDR2 SDRAM Specification.* JEDEC Solid State Tech. Association, May 2006.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[15] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA '81: Proc. of the 8th An. Symp. on Comp. Arch.*, pages 81–87, 1981.

[16] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *ISPASS*, 2001.

[17] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[18] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[19] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[20] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[21] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[22] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[23] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[24] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[25] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Arch. Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[26] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[27] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical report, HP Laboratories Palo Alto, 2006.

[28] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 339–352, 2007.

Appendix C

Paper A.IV

Managing Chip Multiprocessor Memory Systems with Miss Bandwidth Allocations

Magnus Jahre, Marius Grannæs and Lasse Natvig
Submitted to *IEEE Transactions on Computers*
2010

Abstract

Chip Multiprocessors (CMPs) share on-chip units to achieve good resource utilization. This design choice makes destructive interference possible and may cause performance degradations. Resource allocation systems can avoid this problem, and previous approaches have provided separate resource allocation systems for each shared hardware-controlled unit. In this work, we show that resource sharing can be globally controlled by carefully orchestrating the miss bandwidth available to each running process. Our Miss Handling Architecture Bandwidth Control (MHABC) technique improves system performance by tuning the maximum number of concurrent shared memory requests for each process to runtime interference patterns. MHABC leverages a novel interference measurement methodology that estimates the interference-free IPC of a process with an average error of -0.3% and a standard deviation of 12.0%. When MHABC is configured to optimize for the Harmonic Mean of Speedups (HMoS) metric, it improves HMoS by up to 106% and fairness by up to 200% with a worst-case reduction in throughput of 3%.

C.1 Introduction

Chip Multi-Processors (CMPs) commonly share parts of the memory system. This resource sharing is often beneficial since it can lead to improved resource utilization and low-latency interprocessor communication. Unfortunately, the presence of shared resources makes destructive interference possible [27]. Consequently, the performance of an application may be influenced significantly by the applications it is co-scheduled with. This lack of *performance predictability* may be an annoyance to the desktop user, but it can be a business critical issue for data center operators. With the advent of cloud computing, where thousands of distinct users share a common computing infrastructure, on-chip resource allocation may become critical [1].

A considerable research effort has been aimed at improving the way the shared units handle independent, co-executing processes. Efforts have been directed towards the hardware-controlled memory system [3, 13, 20, 27, 37], the shared last-level cache [4, 12, 21, 29, 30, 38–41], the memory bus [11, 25, 26, 28, 33] and system software [9, 32]. Common for these approaches is that all running processes have a static and equal miss bandwidth allocation. In this work, we show that *asymmetric* miss bandwidth allocations can be used to partition memory system resources among processes. We provide a transparent system that adapts per-core miss bandwidth to inter-process interference which we call *Miss Handling Architecture Bandwidth Control (MHABC).*

The subsystems of MHABC and other resource allocation schemes can be divided into three categories: enforcement mechanisms, feedback mechanisms and allocation policies [27]. Our enforcement mechanism leverages that modern caches are non-blocking and potentially support several concurrent cache misses [22]. This feature is provided by the *Miss Handling Architecture (MHA)* and the key component of the MHA is the *Miss Status/Information Holding Registers (MSHRs).* The key observation underlying the design of MHABC is that the number of MSHRs in the last-level private cache can be used to control the number of concurrent requests in the shared memory system. We refer to an MHA where the number of MSHRs can be changed at runtime as a *Dynamic MHA (DMHA)* [15].

MHABC uses the *Dynamic Interference Estimation Framework (DIEF)* [17] to provide interference feedback. DIEF provides measurements of the current memory latency and estimates the memory latency a process would have experienced with exclusive access to all shared resources. Modern memory systems have a considerable amount of parallelism available, and the ability of a process to utilize this parallelism is known as *Memory Level Parallelism (MLP)* [31]. In this work, we combine DIEF's latency estimates with MLP measurements to accurately estimate the performance a process would experience with exclusive access to shared resources. These IPC estimates have an average relative error of -0.3%, -3.5% and -5.8% and a standard deviation of 12.0%, 13.0% and 13.7% for our 4, 8 and 16-core CMPs.

MHABC's allocation policy combines DIEF's latency estimates with a novel miss bandwidth performance model to provide runtime estimates of the chosen performance metric. Based on these estimates, MHABC finds a good bandwidth allocation by efficiently searching through the solution space provided by the model. MHABC supports the Harmonic Mean of Speedups (HMoS) [23], Aggregate Weighted Speedup (AWS) [35], Fairness [8] and Aggregate IPC policy metrics. When MHABC is configured to optimize for the Harmonic Mean of Speedups (HMoS) metric, it improves HMoS by up to 106% and fairness by up to 200% with a worst-case reduction in throughput of 3%. Although we focus on miss bandwidth allocation in this work, the allocation policy is general. It can be applied to any resource allocation mechanism if the latency effect of the allocation can be modeled.

C.2 Background

C.2.1 Interference and Performance Metrics

When evaluating CMP memory system fairness, it is convenient to compare to a baseline where interference does not occur. One way of creating such a baseline is to let the process run in one processing core of the CMP and leave the remaining cores idle [6, 26]. Consequently, the process has exclusive access to all shared resources, and we will refer to this configuration as the *private mode*. Conversely, all processing cores are active and the processes compete for shared resources in the *shared mode*. We define the interference I_p experienced by a processor p as the difference between the shared mode latency L_p and private mode latency $\mathcal{L}_p$ (i.e. $I_p = L_p - \mathcal{L}_p$). This definition is an extension of the interference definition by Mutlu and Moscibroda [25].

A shared mode estimate of a private mode value $\hat{\mathcal{X}}$ may differ from the actual private mode value $\mathcal{X}$. For these estimates to be useful for allocation decisions, it is important that the difference between them is minimized. Consequently, we define the measurement error to be $E = \hat{\mathcal{X}} - \mathcal{X}$. Since latency and cache miss estimates are used for shared mode allocation decisions, we define the relative error $\mathcal{E}^S$ as the absolute error E divided by the shared mode value X ($\mathcal{E}^S = E/X$). For performance measurements, we use the error relative to the private mode $\mathcal{E}^P = E/\mathcal{X}$.

Table C.1 shows the system performance metrics used in this work. Here, P_p and $\mathcal{P}_p$ represent the shared and private mode performance of process p, respectively. Eyerman and Eeckhout [6] showed that the *Aggregate Weighted Speedup (AWS)* [35] and *Harmonic Mean of Speedups (HMoS)* [23] metrics represent system throughput and average normalized turnaround time, respectively. AWS is a system-oriented metric, and HMoS is a user-oriented metric. In addition, we use the fairness metric [8] which measures the difference in shared to private mode slowdown between the running processes. Consequently, fairness is maximized when all processes experience the same slowdown. Finally, we also include the *Aggregate IPC (AI)* metric.

Table C.1: Multiprogrammed Workload Performance Metrics

Metric	Formula	System-Level Meaning [6]	Reference
Aggregate Weighted Speedup (AWS)	$\sum_{p=0}^{n} P_p/\mathcal{P}_p$	System Throughput	Snavely and Tullsen [35]
Harmonic Mean of Speedups (HMoS)	$\frac{n}{\sum_{p=0}^{n} P_p/\mathcal{P}_p}$	Inverse of Average Normalized Turnaround Time	Luo et al. [23]
Fairness	$\frac{min(P_i/\mathcal{P}_i)}{max(P_j/\mathcal{P}_j)}\ i,j \in \{0,n\}$	Assumed by system software	Gabor et al. [8]
Aggregate IPC (AI)	$\sum_{p=0}^{n} P_p$	None [6]	-

This metric values high IPC numbers and the best performance is achieved by maximizing the IPC of the high IPC processes. For this reason, it is not recommended to use Aggregate IPC as a performance metric [23, 35], and we only include it to illustrate the effects of not using private mode performance in allocation decisions.

C.2.2 Modern Memory Bus Interfaces

Memory bus scheduling is a challenging problem due to the 3D structure of DRAM consisting of rows, columns and banks. Commonly, a DRAM read transaction consists of first sending the row address, then the column address and finally receiving the data. When a row is accessed, its contents are stored in a register known as the row buffer, and a row is often referred to as a *page*. If the row has to be activated before it can be read, the access is referred to as a *row miss* or *page miss*. It is possible to carry out repeated column accesses to an open page, called *row hits* or *page hits*. This is a great advantage as the latency of a row hit is much lower than the latency of a row miss. The situation where two consecutive requests access the same bank but different rows is known as a *row conflict* and is very expensive in terms of latency. DRAM accesses are pipelined, so there are no idle cycles on the memory bus if the next column command is sent while the data transfer is in progress. Furthermore, command accesses to one bank can be overlapped with data transfers from a different bank.

Rixner et al. [34] proposed the First Ready - First Come First Served (FR-FCFS) algorithm for scheduling DRAM requests. FR-FCFS reorders memory requests to achieve high page hit rates which result in increased memory bus utilization. This algorithm prioritizes requests according to three rules: prioritize ready commands over commands that are not ready, prioritize column commands over other commands and prioritize the oldest request over younger requests.

C.2.3 Miss Handling Architectures (MHAs)

A generic MHA consists of n MSHRs which store the cache block address of the miss, target information and a valid bit (see Figure B.1 on page 109). The cache can support as many concurrent misses to different cache blocks as there are MSHRs. Each MSHR commonly has its own comparator and the MHA can be described as a small fully associative cache. For each miss, the information required for the cache to answer the processor's request is stored. This target information determine the number of misses to the same cache block that can be handled without blocking [7, 22]. The cache must block when all valid bits are set, and a blocked cache cannot service any requests.

C.3 A Miss Bandwidth Allocation Model

To optimize system performance by changing miss bandwidth allocations, it is helpful to develop a model that describes the performance impact of bandwidth allocations. We base our model on the model by Karkhanis and Smith [19]. The underlying idea is that performance can be modeled by quantifying the steady-state performance (i.e. perfect branch predictor and perfect caches) and then subtracting the performance loss due to cache misses and branch mispredictions.

$$P_p = \mathrm{IPC}_p = \frac{N_p}{C_p^{\mathrm{Compute}} + C_p^{\mathrm{MemStall}}} \tag{C.1}$$

Equation C.1 expresses the performance P_p of processor p as a function of the number of committed instructions N_p, the number of clock cycles used for computation C_p^{Compute} and the number of clock cycles where the processor is stalled waiting for memory requests C_p^{MemStall}. N_p and C_p^{Compute} mainly depend on the characteristics of the running process and the processor implementation and are therefore independent of miss bandwidth. Consequently, changing the miss bandwidth allocation will mainly affect C_p^{MemStall}. Branch mispredict stalls are part of C_p^{Compute} in our model since it is unlikely that changing miss bandwidth allocations will affect the branch misprediction rate significantly.

Figure C.1 illustrates our performance model. First, the process is running at its steady-state IPC before it experiences a cache miss. Then, IPC is reduced while the instructions that do not depend on the pending load are serviced. This results in two additional cache misses. Finally, only instructions that depend on the three loads are left in the reorder buffer and the processor stalls. Some time later, the requested data is received from memory. Then, the processor starts committing instructions again and the IPC ramps up to its steady-state value.

In this example, the latencies of the two additional cache misses were hidden by the first cache miss. Consequently, the relationship between the average memory

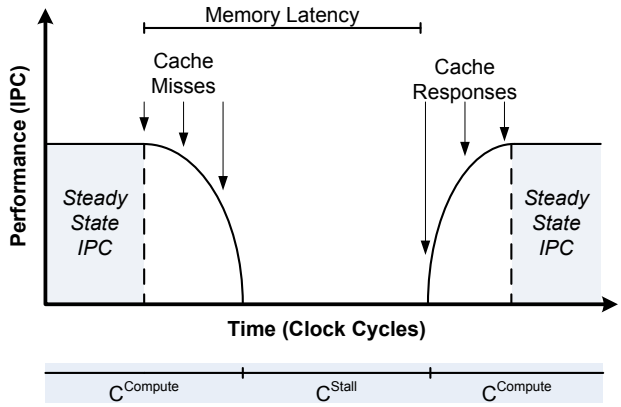


Figure C.1: Memory Level Parallelism Example

latency L_p and processor stall time due to memory C_p^{MemStall} depends on the ability of the process to utilize the parallelism available in the memory system. This ability is commonly referred to as *Memory Level Parallelism (MLP)* [31].

$$C_p^{\text{MemStall}} = M_p \cdot L_p \tag{C.2}$$

Equation C.2 relates C_p^{MemStall} to L_p and average MLP M_p. In this work, we improve performance by adapting the miss bandwidth available to each process to runtime interference patterns. Consequently, M_p is a function of the available bandwidth b_p, and b_p can be controlled by manipulating the number of MSHRs available in the last-level private cache. Reducing b_p will often reduce the performance of process p. However, it may increase the performance of one or more of the other processes by reducing their average shared memory latencies. Consequently, we extend Equation C.2 by making both M_p and L_p functions of the current bandwidth allocations:

$$C_p^{\text{MemStall}} = M_p(b_p) \cdot L_p(b_1, b_2, \ldots, b_n) \tag{C.3}$$

Similarly, the private mode stall time $\mathcal{C}_p^{\text{MemStall}}$ can be described by the following equation:

$$\mathcal{C}_p^{\text{MemStall}} = \mathcal{M}_p(b_p^{\max}) \cdot \mathcal{L}_p(b_p^{\max}) \tag{C.4}$$

The main component of most CMP performance metrics (see Table C.1) is the speedup of shared mode performance relative to private mode performance [6]. To ensure that this inter-mode comparison is valid, we need to compare the same

Table C.2: Variable Summary

Shared Mode	Private Mode	Description
P	$\mathcal{P}$	Performance (IPC)
n	-	Number of processing cores
N	$\mathcal{N}$	Number of committed instructions
C^{Compute}	$\mathcal{C}^{\text{Compute}}$	Clock cycles used for computation
C^{MemStall}	$\mathcal{C}^{\text{MemStall}}$	Clock cycles stalled waiting for memory
M	$\mathcal{M}$	Memory Level Parallelism (MLP)
L	$\mathcal{L}$	Total shared memory system latency
b	-	Miss bandwidth allocation
I	-	Total interference clock cycles
R	-	Number of memory requests (i.e. loads, stores and writebacks)
S	-	Number of memory loads and stores
F	-	Number of free bus slots
A	-	Additional bus requests

instruction sequence from both modes. Consequently, the number of instructions in the private mode $\mathcal{N}$ is equal to its shared mode counterpart N. By substituting Equations C.3 and C.4 into Equation C.1 and exploiting that N is equal to $\mathcal{N}$, we can express the shared to private mode speedup with the following equation:

$$\frac{P_p}{\mathcal{P}_p} = \frac{\mathcal{C}_p^{\text{Compute}} + \mathcal{M}_p(b_p^{\max}) \cdot \mathcal{L}_p(b_p^{\max})}{C_p^{\text{Compute}} + M_p(b_p) \cdot L_p(b_1, b_2, \ldots, b_n)} \tag{C.5}$$

Because of the similarities between the private and shared modes, it is possible to simplify Equation C.5 further. Firstly, MLP quantifies the ability of a process to make use of the parallelism available in the memory system. Consequently, MLP mainly depends on the characteristics of the process and the processor core implementation. This observation enables the simplifying assumption that M_p is approximately equal to $\mathcal{M}_p$. Secondly, C_p^{Compute} is by definition independent of memory latency. Therefore, it can be used interchangeably with its private mode counterpart $\mathcal{C}_p^{\text{Compute}}$.

Equation C.2 states that if we measure two out of M_p, L_p and C_p^{MemStall} we can compute the last one. In this work, we choose to estimate L_p and C_p^{MemStall} and compute M_p. For convenience, Table C.2 summarizes the variables used in our models.

C.4 Estimating the Effects of Bandwidth Allocation Changes

Figure C.2 illustrates the high-level operation of our miss bandwidth allocation technique. First, we divide time into fixed-size periods. Then, we collect measurements during an *observation period* where all processes have their maximum

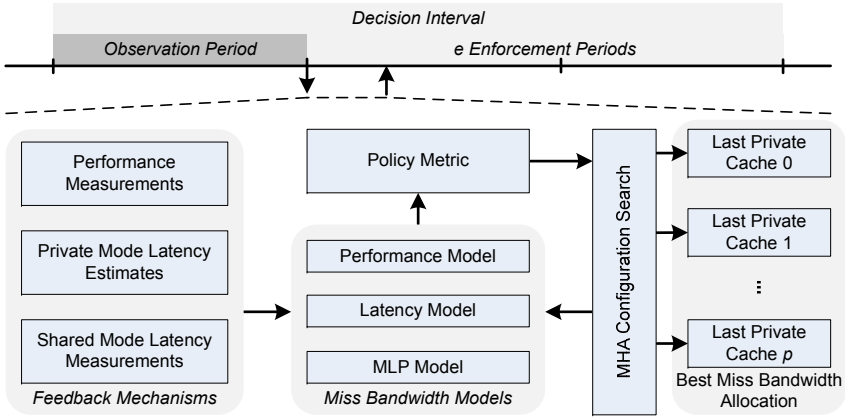


Figure C.2: Miss Bandwidth Allocation Flow

Algorithm 3 Online Performance Metric Estimation

```
procedure EvaluateBandwidthAllocation(b_1, ..., b_n)
    for Processor p do
        if (R_p < T_r or σ_{R_p}/μ_{R_p} > T_v) and b_p < b^max then
            return lowest metric value
        end if
    end for
    L'_1, ..., L'_n = EstimateSharedLatencies(b_1, ..., b_n)
    for Processor p do
        M'_p = EstimateMLP(b_p)                          ▷ Equation C.12
        P'_p = EstimatePerformance(M'_p, L'_p)           ▷ Equation C.13
    end for
    return ComputeMetricValue(P'_1, ..., P'_p, P'_1, ..., P'_p)
end procedure
```

bandwidth allocation. This is necessary to gather accurate MLP measurements. MHA Configuration Search then estimates the value of a policy metric for various bandwidth allocations. The policy metric can be any of the system performance metrics in Table C.1. Finally, we enforce the best performing miss bandwidth allocation by changing the number of available MSHRs in the last-level private cache of each core. This allocation is used for the next e *enforcement periods*.

In this section, we discuss the miss bandwidth models while the mechanisms and search algorithms are discussed in Section C.5. Algorithm 3 is the top-level miss bandwidth performance estimation model, and the rest of this section is devoted to explaining it in detail. The main part of the algorithm is the procedures used to estimate the new MLP M'_p and new shared latency L'_p. We discuss our L'_p estimation model in Section C.4.1 and our M'_p estimation model in Section C.4.2. In Section C.4.3, we combine L'_p and M'_p with the performance model from Section C.3 to estimate the performance of a process with a certain miss bandwidth allocation.

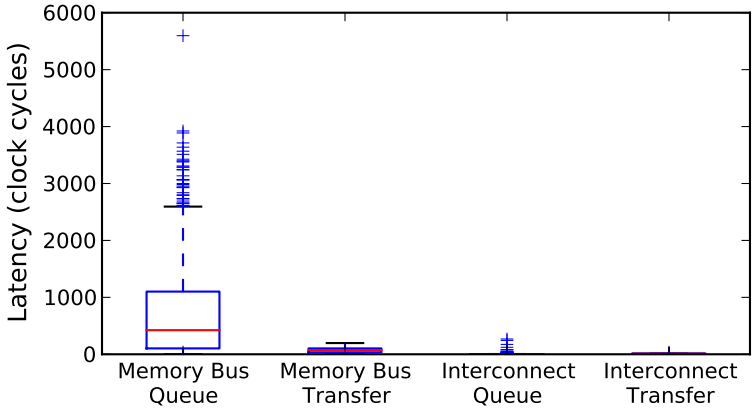


Figure C.3: Shared Mode Latency Variation

Our miss bandwidth allocation technique assumes that the past predicts the future. This assumption is unlikely to hold if there are large variations in the number of requests R_p between periods. To avoid making predictions in these cases, we use the e enforcement periods to sample R_p. From these samples, we estimate the standard deviation σ_{R_p} of R_p. If the ratio of the standard deviation to the mean of R (i.e. σ_{R_p}/μ_{R_p}) is greater than the acceptable variation threshold T_v, we refrain from making predictions. Furthermore, DIEF needs a certain number of requests for the private latency estimates to be accurate. Therefore, we require that the number of requests must be larger than the request threshold T_r.

C.4.1 Shared Memory Latency Estimation

A large part of the latency in the shared memory system is due to fixed latency operations like cache accesses and data transfers. Consequently, the latency change from miss bandwidth reductions will mainly affect queuing latencies. In our memory system, a request can be queued in the on-chip interconnect and the memory bus. Figure C.3 visualizes per-unit average latencies for our 4-core CMP with a box plot. Here, the upper and lower edges of the box are the upper and lower quartiles of the data, and the line is the median. The whiskers show the maximum and minimum value that is within a factor of 1.5 of the interquartile range. Values outside this range (outliers) are shown as crosses. Figure C.3 shows that the memory bus queue is the main source of latency variation. This is not surprising as a single bus transaction takes between 40 and 260 processor cycles in our model which dwarfs the latencies of the interconnect queue.

This observation makes is possible to simplify the estimation task to only estimating the new memory bus latency, and we discuss our bus latency model in Section C.4.1.1. However, some of the memory bus latency is caused by shared

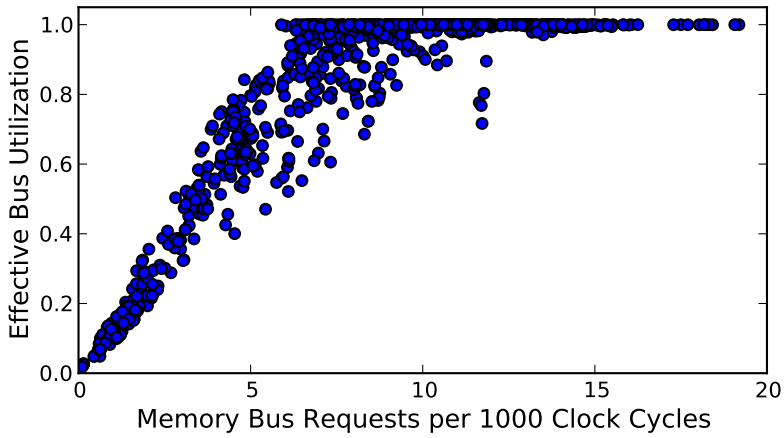


Figure C.4: Effective Memory Bus Utilization

cache interference misses. Reducing the bandwidth allocation for one process will reduce its shared cache access frequency and thus its ability to evict the cache blocks of other processes. This indirect effect can have large performance implications as it can remove a considerable part of the memory latency for processes that are exposed to severe cache interference. We describe our cache interference model in Section C.4.1.2. Finally, Section C.4.1.3 leverages the cache and bus models to provide a miss bandwidth allocation aware latency model for the complete hardware-managed, shared memory system.

C.4.1.1 Estimating Memory Bus Latency Change

Our memory bus queue model is based on the observation that access control is necessary when the memory bus is running at its full effective capacity. This effective capacity is commonly less than the theoretical maximum capacity. The reason is that the commands necessary to carry out a data transfer can only be partially overlapped with other data transfers. The ability to overlap commands and data transfers depend on the page locality of the running processes. Consequently, the effective memory bus capacity can vary considerably between workloads.

To gain insight into this behavior, we define the term *effective utilization* which is the number of cycles the bus is busy serving requests (both commands and data transfers) over the total number of cycles. Figure C.4 plots the effective memory bus utilization and the memory bus request intensity (i.e. requests per 1000 clock cycles) for our 4-core CMP model. There are two main cases. When the memory bus is working at less than its effective capacity, effective utilization scales roughly linearly with the request intensity. However, the effective utilization can vary significantly for a given request intensity. In the other case, the memory bus is

Algorithm 4 Shared Memory Latency Estimation

procedure EstimateSharedLatencies($b_0, \ldots, b_n$)
  Initialize $A_0, \ldots, A_n = 0$
  Compute F ▷ Equation C.7
  for p from 0 to n **do**
    if $S_p^{\text{Bus}} > T_r^{\text{Bus}}$ and $b_p == b^{\max}$ **then**
      Compute A_p ▷ Equation C.8
    end if
  end for
  if EffectiveBusUtilization $> T_B$ and $F > 0$ **then**
    Compute R' ▷ Equation C.9
  else
    $R'_p = R_p \cdot \text{MRR}$ for $p \in \{0, n\}$
  end if
  for p from 0 to n **do**
    if $b_p == b^{\max}$ **then**
      Compute $L_p^{\text{CacheMiss}'}$ ▷ Equation C.10
    else
      $L_p^{\text{CacheMiss}'} = 0$
    end if
  end for
  for p from 0 to n **do**
    Compute L'_p ▷ Equation C.11
  end for
end procedure

working at its full effective capacity. Here, the effective bus utilization is close to one for a range of request intensities.

Based on these observations, we focus our efforts towards modeling the effects of bandwidth redistribution when the memory bus saturates. Our model divides the capacity of the memory bus into n fixed size slots with a length equal to the average memory bus transfer latency. Reducing the MLP of one or more of the running processes will then result in freeing up a certain number of slots F. Processes that are sensitive to memory latency, will increase their performance and occupy some of these slots. The combination of these trends will determine the new average shared latencies. Algorithm 4 outlines this procedure more formally, and the rest of this section will be used to explain the algorithm in detail.

$$\text{MRR} = \frac{\hat{M}_p(b_p^{\max})}{\hat{M}_p(b_p^{\text{new}})} = \frac{\bar{R}_p^{\text{Concurrent}}(b_p^{\text{new}})}{\bar{R}_p^{\text{Concurrent}}(b_p^{\max})} \tag{C.6}$$

Equation C.6 introduces the *MLP Reduction Ratio (MRR)*. MRR estimates the percentage of the concurrent requests that will be available with the new bandwidth allocation b_p^{new} and is the central component of our latency estimation technique. Qureshi et al. [31] defined *mlp-cost* as the average of the inverse of the number of concurrently serviced memory requests. Consequently, the mlp-cost $\hat{M}_p$ is a number between $\frac{1}{16}$ and 1 if we assume 16 MSHRs in the last-level private cache. If a process has an mlp-cost of 1, no memory accesses are overlapped. Conversely,

an mlp-cost of $\frac{1}{16}$ signifies that there are always 16 concurrent memory request. Since we use MRR to scale request measurements, we need to divide $\hat{M}_p(b_p^{\max})$ by $\hat{M}_p(b_p^{\text{new}})$ for the ratio to be proportional to the memory request change.

$$F = \sum_{p=0}^{n} R_p^{\text{Bus}} \cdot (1 - \text{MRR}) \tag{C.7}$$

Equation C.7 estimates the number of bus slots that are made available with a new bandwidth allocation. If we multiply MRR with the number of bus requests, we will get an estimate of the number of bus requests with the new bandwidth allocation. However, we need the number of *removed* requests for our estimation procedure. To achieve this, we multiply R_p^{Bus} by the complement of MRR (i.e. the percentage of concurrent requests that are removed).

$$A_p = S_p^{\text{Bus}} \cdot \frac{L_p^{\text{bq}}}{\hat{\mathcal{L}}_p^{\text{bq}}} \tag{C.8}$$

A bandwidth sensitive process will use the bandwidth made available by the bandwidth reduction of others to increase its performance. This behavior results in an increase in the number of requests per sample, and Equation C.8 shows how we model it. Here, A_p is the number of additional bus requests. We estimate the bandwidth need of an application by the ratio of its shared mode bus queue latency L_p^{bq} to its estimated private mode bus queue latency $\hat{\mathcal{L}}_p^{\text{bq}}$. The idea is that a large difference in bus queue latencies between the private and shared modes is an indication that the application can put memory bandwidth to good use. Conversely, it is unlikely that a process with similar bus queue latencies can make use of the additional bandwidth.

We only run the A_p estimation for processes that have not had their miss bandwidth decreased (see Algorithm 4). This is based on the assumption that a miss bandwidth reduction reduces the ability of a process to increase its request count. Furthermore, we assume that a process will not increase its number of requests if it has fewer than T_r requests in the sample.

$$R'_p = \begin{cases} R_p \cdot \text{MRR} + \left(F \cdot \frac{A_p}{\sum_{i=0}^{n} A_i}\right) & \sum_{p=0}^{n} A_p > F \\ R_p \cdot \text{MRR} + A_p & \text{otherwise} \end{cases} \tag{C.9}$$

Equation C.9 estimates the number of memory requests R'_p this sample would have contained with the new bandwidth allocation. Since the memory bus has finite capacity, Equation C.9 consists of two cases. In the first case, the number additional of bus requests A_p is larger than the number of free bus slots F. Here, we assume that the running processes will distribute the free bus slots F among themselves according to their estimated bandwidth needs. In the other case, the

number of additional bus requests A_p is less than F and we can use A_p directly. We only use Equation C.9 if the effective bus utilization is greater than the bus utilization threshold T_B and if there are free bus slots (see Algorithm 4).

C.4.1.2 Estimating Cache Interference Reduction

A reduction in miss bandwidth leads to a reduction in shared cache access frequency. Consequently, cache capacity interference is also reduced since the new allocation degrades the ability of a process to evict other processes' cache blocks. The latency reduction from this effect may be large as it has the potential of removing a large part of the memory bus latency.

$$\begin{aligned} I_p^{\text{Cache}'} &= I_p^{\text{Cache}} \cdot \left(1 - \frac{\sum_{p=0}^{n} R_p \cdot \text{MRR}}{\sum_{p=0}^{n} R_p}\right) \\ &= I_p^{\text{Cache}} \cdot (1 - \text{CAF}) \end{aligned} \tag{C.10}$$

Equation C.10 estimates the saved cache interference latency. The main assumption is that the reduction in cache interference latency is proportional to the average reduction in access frequency. To quantify this notion, we define the *Cache Access Frequency (CAF)* metric. CAF is the estimated number of cache requests in one observation period with a certain miss bandwidth allocation. Since we are interested in the *reduction* in cache interference, we use the complement of CAF to scale the measured cache interference latency I_p^{Cache}.

C.4.1.3 The Shared Latency Model

$$L'_p = \left(L_p - L_p^{\text{BusQueue}}\right) + \left(L_p^{\text{BusQueue}} \cdot \frac{R_p}{R'_p}\right) - I_p^{\text{Cache}'} \tag{C.11}$$

Equation C.11 combines the memory bus queue and cache interference estimates to provide a shared memory system latency estimate L'_p. First, we compute the miss bandwidth independent part of the shared memory latency by subtracting the average memory bus queue latency component L_p^{BusQueue} from the complete average shared memory latency L_p. Then, we compute the new memory bus queue latency estimate by scaling the memory bus queue measurement with the estimated change in request count. Finally, we subtract the estimated latency savings from the reduction in cache interference.

In this work, we investigate a hardware realization of dynamic access control. Consequently, there are a number of possible implementations of Equation C.11 with different area- and latency requirements. Since we use the selected MHA for a large number of cycles, we can afford a high-latency serial implementation. In fact, an ALU, a multiplier and a division unit are sufficient to compute Equation C.11. This

Table C.3: L'_p Computation Latency

Unit	Worst Case #Operations	Unit Latency	Total Latency
Add	6	1	6
Subtract	4	1	4
Multiply	6	3	18
Divide	5	20	100
Total			**128**

results in a total evaluation latency of 512, 1024 and 2048 cycles for the 4-, 8- and 16-core CMPs, respectively (see Table C.3 for details). In this analysis, control flow is not taken into account. This is not a problem since we will show that MHABC is not sensitive to small estimation latency variations (Section C.7.1.3).

C.4.2 Estimating Memory Level Parallelism Change

$$M'_p = M_p(b_p^{\max}) \cdot \frac{\hat{M}(b_p^{\text{new}})}{\hat{M}(b_p^{\max})} \tag{C.12}$$

We model processor MLP change as shown in Equation C.12. The main observation underlying this equation is that the available MLP is different when viewed by the processor core and last level private cache [42]. Therefore, we compute the new processor MLP M'_p by scaling $M_p(b_p^{\max})$ with the percentage reduction in mlp-cost.

C.4.3 Estimating Memory Stall Time

$$P'_p = \frac{N_p}{C_p^{\text{Compute}} + C_p^{\text{MemStall}'}} = \frac{N_p}{C_p^{\text{Compute}} + M'_p \cdot L'_p} \tag{C.13}$$

Equation C.13 estimates performance P'_p with the shared latency estimate L'_p and the MLP estimate M'_p. Here, we leverage the observation that miss bandwidth allocations only influence performance through changing the number of cycles the processor is stalled waiting for memory ($C_p^{\text{MemStall}'}$).

C.5 MHABC - A Practical Miss Bandwidth Allocation System

This section introduces our *Miss Handling Architecture Bandwidth Control (MHABC)* technique. MHABC is a complete resource allocation implementation for the hardware-controlled shared memory system and implements the models discussed in Section C.3 and C.4. Figure C.5 shows high-level structure of MHABC. The last-level private caches implement the *Dynamic Miss Handling Architecture (DMHA)*

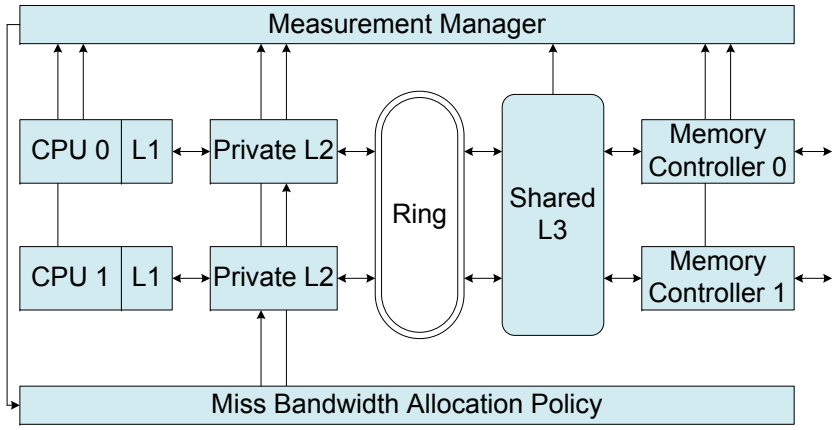


Figure C.5: Miss Handling Architecture Bandwidth Control (MHABC) System Architecture

allocation mechanism [14, 15]. Furthermore, the on-chip interconnect, the shared L3 cache and the memory controllers have been extended to support the *Dynamic Interference Estimation Framework (DIEF)* [17]. DIEF provides highly accurate estimates of the private mode average memory latency through the shared memory system as well as shared mode latency measurements. In addition, we add hardware for measuring processor stall cycles due to memory and last-level private cache MLP. The *Measurement Manager* is responsible for collecting and aggregating measurements. These measurements are then provided to the *Miss Bandwidth Allocation Policy (MBAP)*. The MBAP uses the performance model from Section C.3 and the estimation model from Section C.4 to find the allocation that maximizes a given performance metric. This allocation is then enforced by the DMHA allocation mechanism.

C.5.1 The DMHA Allocation Mechanism

Figure C.6 illustrates the modifications to a conventional *Miss Handling Architecture (MHA)* that are needed to enable runtime miss bandwidth allocations [14, 15]. Here, the conventional MHA is augmented with one usable bit U for each MSHR and a control unit. When the usable bit is set, the MSHR is allowed to store miss data. If all usable MSHRs are occupied, the cache must block. Consequently, DMHA makes it possible to control the number of concurrent shared memory system accesses.

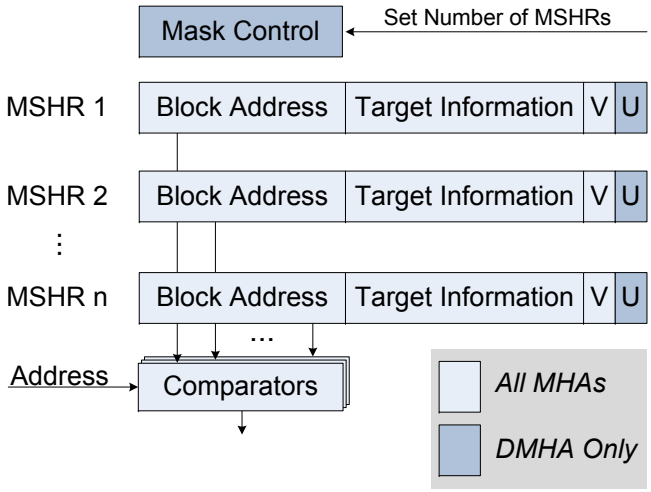


Figure C.6: Dynamic Miss Handling Architecture (DMHA)

C.5.2 The Feedback Mechanisms

This section describes the feedback mechanisms used in MHABC. The most complex mechanisms are devoted to estimating private mode latencies and are part of the *Dynamic Interference Estimation Framework (DIEF)* [17]. In particular, we use the memory bus, shared cache and interconnect mechanisms from DIEF. Furthermore, we need measurements of MLP and processor memory stall cycles. The storage overheads of the feedback mechanisms are summarized in Table C.4.

C.5.2.1 Estimating Memory Bus Interference

Memory bus interference estimation is complicated since modern memory controllers aggressively reorder requests to achieve good page hit rates and high bus utilization. Consequently, the latency of a memory request depends on the other requests serviced in close temporal proximity. The estimation task is further complicated by the presence of shared mode only cache misses and shared cache write-backs.

Fortunately, a core's access order is very similar in the two modes. Consequently, the private mode latency can be estimated by adding hardware that emulates private mode memory bus scheduling at runtime. We allocate a circular buffer to store the request addresses in arrival order. In addition, we use B registers to store the active page address for each memory bank (where B is the number of memory banks). This makes it possible to estimate if a request is a private mode

Table C.4: Feedback Measurement Storage Requirements

	Storage Requirements (KB)		
	4-core	**8-core**	**16-core**
Memory Bus Interference	2.3	9.2	18.4
Memory Bus Latency	0.25	0.5	0.5
Auxiliary Tag Directories	10.5	20.0	38.0
Cache Interference Latency	0.25	0.5	1.0
Interconnect Latency and Interference	0.03	0.06	0.13
MLP Estimation	0.27	0.53	1.06
CPU Stall Time Measurement	0.03	0.06	0.13
Measurement Manager	0.17	0.34	0.69
Total	**13.8**	**31.2**	**59.9**

page hit, miss or conflict. To estimate transfer and queue latencies, we emulate the FR-FCFS scheduling algorithm [34]. For each serviced request, we use a pointer to store the execution order. We estimate the latency of each element with a lookup table since only a few of the possible DRAM latencies occur frequently in practice [17]. By storing a pointer to the oldest pending request at the time of insertion, we can estimate the queue latency by traversing the execution order and accumulating transfer latencies. We refer the reader to Jahre et al. [17] for additional details.

If the memory bus queue is full, a request will be delayed in the shared cache MSHRs. We refer to this latency as a memory bus entry latency. In the private mode, the number of concurrent memory requests are limited by the number of MSHRs in the private caches. This number of MSHRs is commonly considerably less than the number of entries in the memory bus queue since the bus queue is dimensioned to handle requests from n processors. Consequently, we can assume that all entry latencies are interference.

The storage overhead of the memory bus interference estimation technique scales with the number of cores and the number of memory bus channels. The storage overhead for one core and one channel is 4706 bits [15]. In addition, we need to measure the shared mode memory bus latency. For this, we allocate a 32 bit counter for each of the 64 read queue entries. This value only scales with the number of channels since only one request can occupy a queue entry at the time.

C.5.2.2 Estimating Cache Capacity Interference

To estimate the number of misses a process would experience with exclusive access to the shared cache, we use one Auxiliary Tag Directory (ATD) per core [5, 31]. The ATDs require significantly less storage than the full cache as they only store the tag for each cache block and not its data. All shared cache requests from a process is directed to one of the ATDs. Consequently, an interference miss is a request that hits in the ATD and misses in shared cache. We measure the latency of the interference misses by allocating a 32 bit counter for each of the shared cache MSHRs.

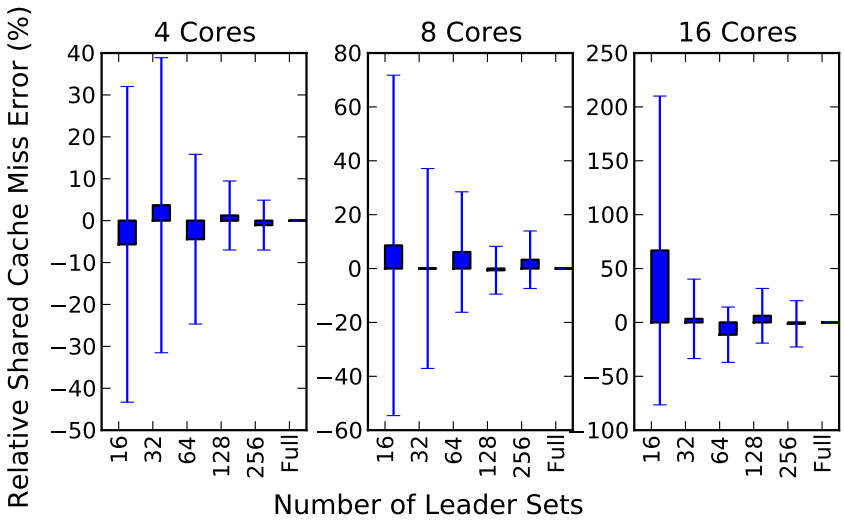


Figure C.7: Shared Cache Miss Estimation Error

The original DIEF proposal used complete tag directories to achieve high accuracy [17]. This choice has a storage cost that we would like to avoid for MHABC. To achieve this, we employ set sampling [5, 31]. Here, the cache tags are divided into *n constituencies*. Then, one *leader set* is chosen for each constituency and the remainder of the sets are designated *follower sets*. Only the leader sets are stored in the ATD and we assume that the events that occur in this set also will occur in the follower sets. We use Qureshi et al.'s simple-static policy for selecting leader sets [31].

The memory bus interference feedback mechanism requires that the cache interference mechanism informs it of which requests are interference misses and about private mode shared cache writebacks. With a full ATD, these requirements are easy to meet as we can detect events with a per set granularity. To meet these requirements with set sampling, we assume that an interference miss in a leader set indicates interference misses for all follower sets. We implement this by adding a counter that is incremented by the number of sets in a constituency for each detected interference miss. If this counter is larger than 0, we tag the cache miss as an interference miss and decrement the counter. We use the same policy to insert private mode only writebacks.

Figure C.7 shows the relative errors with one standard deviation for shared cache miss estimates and simulation point 0. We remove benchmarks with less than 100 shared mode misses to avoid that they skew the results. A full-map ATD is very accurate and the variability increases steadily with more aggressive sampling. However, the impact on the overall latency from these errors are low. Therefore, we choose 64 leader sets for all our CMPs to achieve a good trade-off between accuracy and storage overhead.

Algorithm 5 Memory Level Parallelism Estimation

```
procedure MLPEstimation
    a = Number of allocated MSHRs
    if a > 0 then
        for Number of MSHRs m from 1 to b_max do
            if m < a then
                MLPCost[m] += 1/m
            else
                MLPCost[m] += 1/a
            end if
        end foractiveCycles += 1
    end if
end procedure
```

C.5.2.3 Estimating Interconnect Interference

Interconnect interference occur when a request belonging to one core delays requests from a different core. Since interconnect latency is independent of the order of memory requests, it is sufficient to detect this situation and add the additional queue latency for each of the queued requests. Furthermore, the transfer latency can be different in the two modes if the process is scheduled on different cores. We avoid this problem by assuming that the process runs on the same core in the two modes. Finally, all interconnect entry latencies are assumed to be interference since the buffers are dimensioned to handle the request load of n concurrent processes. These measurements require two registers for each processor, one for measuring latencies and one for the interference estimate.

C.5.2.4 Estimating MLP

Our mlp-cost estimator is inspired by Qureshi et al. [31]. Qureshi et al. used their estimator to determine the mlp-cost of cache misses and prioritize cache space for blocks that are costly to retrieve from memory. For MHABC, we need to determine the variation in MLP with different bandwidth allocations. Our solution is to allocate one register for each possible bandwidth allocation in the last-level private cache and run the procedure outlined in Algorithm 5. Conceptually, Algorithm 5 runs every cycle. At the end of each observation period, average MLP is computed by dividing the sum in each register by the number of cycles with outstanding misses. If we assume a 32 bit fixed-point representation, a 32 bit activity counter and 16 different MHAs, the storage overhead of this estimator is 544 bit per core.

C.5.2.5 Measuring CPU Stall Time

To estimate performance we need to quantify MLP from the point of view of the processor core. We opt to do this by measuring the number of clock cycles the processor is stalled waiting for a memory request. Consequently, we add a 32 bit

Algorithm 6 Decomposed Search

```
procedure DecomposedSearch(Bus Requests r_i to r_n)
    o = Sort r in descending order
    bestMHA = b_max, b_max, ..., b_max
    Initialize bestMetricValue
    for Processor p in o do
        for m in Available MSHR Counts do
            currentMHA = copy(bestMHA), currentMHA[p] = m
            v = EvaluateBWAllocation(currentMHA)
            if v >= bestMetricValue then
                bestMHA[p] = m, bestMetricValue = v
            end if
        end for
    end for
    return bestMHA
end procedure
```

counter to each core. This counter is incremented every cycle the processor does not commit instructions. When the counter value is increased beyond a threshold, we assume that the stall is due to a last-level private cache miss. This is based on the observation that the latency to retrieve a cache block from the last-level private cache is higher than other events that may cause stalls (e.g. branch mispredicts). Consequently, the threshold must be greater than the last-level private cache hit roundtrip latency and less than the minimum shared cache hit roundtrip latency. When the processor starts committing instructions again, the value is added to a 32 bit register if it is larger than the threshold. Then, the counter is set to 0.

C.5.3 Allocation Policies

Our allocation policies are based on the observation that most system performance metrics are functions of speedups computed by dividing shared mode performance by private mode performance [6]. Given our performance estimations, we can then compute these metrics at runtime (Equation C.5). Furthermore, our miss bandwidth models make it possible to estimate the performance metric value for a range of miss bandwidth allocations. Consequently, finding the best asymmetric MHA becomes a search for the maximum metric value across all possible MHA configurations. We will refer to the algorithm that exhaustively evaluates all possible MHAs as the *Exhaustive Search*.

Estimating performance with m different MHAs in each private cache and p processors requires m^p MHA estimations. Consequently, the exhaustive search algorithm scales poorly with respect to the number of processors. To improve scalability, we decompose the search space by evaluating each processor independently of the others and keeping the best MHA. Algorithm 6 illustrates this approach. First, we order the processors according to the number of bus requests. This is based on the assumption that the process with the most bus requests has the largest impact on the latency of the other processors. We refer to this algorithm as the *Decomposed*

Table C.5: CMP Model Parameters (4- /8- /16-core)

Parameter	Value
Feature Size	65/45/32 nm
Clock frequency	4 GHz
Processor Cores	128 entry reorder buffer, 32 entry load/store queue, 64 entry instruction queue, 4 instructions/cycle, 4 integer ALUs, 2 integer multiply/divide, 4 FP ALUs, 2 FP multiply/divide, hybrid branch predictor with 2048 local history registers, 4-way and a 2048 entry BTB
L1 Data Cache	2-way, 64KB, 3/2/2 cycles latency, 2 cycles cycle time, 16 MSHRs
L1 Inst. Cache	2-way, 64KB, 3/2/2 cycles latency, 2 cycles cycle time, 16 MSHRs
L2 Private Cache	4-way, 1 MB, 9/6/5 cycles latency, 4/3/2 cycles cycle time, 16 MSHRs
L3 Shared Cache	16-way, 8/16/32 MB, 16/12/12 cycles latency, 4 cycles cycle time, 16/32/64 MSHRs, 4 banks
Ring Interconnect	4/4/8 cycles per hop transfer latency, 1/1/2 pipe stages per hop, 32 entry request queue, 1/2/2 request rings, 1 response ring
Main memory	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [34], Open page policy, 1/2/2 channels

Search, and it requires $m \cdot p$ MHA evaluations. Finally, we reduce the search space further by only considering MHAs where the number of MSHRs is a power of two. This algorithm is called the *Log-Decomposed Search*, and it requires $(\log_2 m + 1) \cdot p$ MHA estimations.

C.6 Methodology

We use the system call emulation mode of the cycle-accurate M5 simulator [2] for our experiments and have extended M5 with a ring interconnect as well as a detailed DDR2-800 memory bus and DRAM model [18]. The parameters of our CMP models can be found in Table C.5.

To evaluate our proposals, we use benchmarks from the SPEC 2000 benchmark suite [36]. We use the SimPoint methodology [10] to choose at most 6 25 million instruction samples that together represent full program execution. Then, we create workloads by randomly selecting benchmarks from the entire SPEC 2000 suite. The only requirements given to the random selection process are that a benchmark can only appear once in each workload and that all benchmarks must appear in at least one workload for each number of processors. The resulting workloads are available in [16].

This process leaves us with 6 simulation points per benchmark and a number of workloads consisting of different benchmarks. To combine these two, we assign an ID to each simulation point. Then, we create one workload for each simulation point ID by combining the simulation points for this ID for all benchmarks in each workload. If a benchmark needs less than 6 simulation points to represent its entire execution, we start over at simulation point 0 for this benchmark. In multiprogrammed mode, we dump the statistics when a benchmark has committed 25 million instructions and continue simulation until all benchmarks in the workload

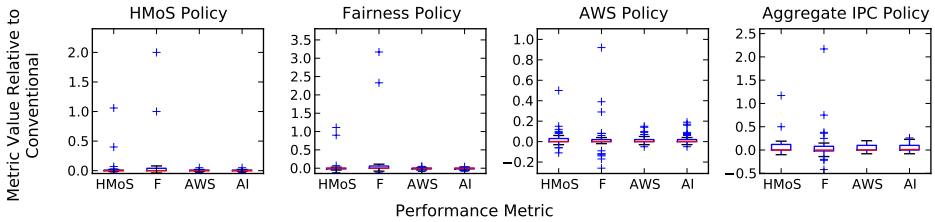


Figure C.8: MHABC 4-core Performance

have committed this number of instructions. If a benchmark terminates before all benchmarks have committed enough instructions, we restart the benchmark at the beginning of its simulation point.

We provide performance measurements for a benchmark by taking the result from each simulation point and combining them using the weight assigned by the SimPoint tool. Consequently, our performance measurements reflect 150 million committed instructions for each benchmark. When we report a system performance metric from multiple simulation points, we compute the representative performance of the benchmark first and then use it in the metric. Due to heavy competition for memory bandwidth, the number of clock cycles used to collect a 25 million instruction sample can be on the order of 7 billion.

C.7 Results

In this section, we provide the results from our experiments with MHABC. MHABC uses a number of thresholds. We set the request threshold T_r to 512, the bus request threshold T_r^{Bus} to 256 and the bus utilization threshold T_B to 0.9375 for easy computation. T_r and T_r^{Bus} can be computed by logical shifts and then checking if the resulting number is larger than 0 while T_B can be checked by testing if the 4 most significant bits of a 20 bit counter are 1. We use a variation threshold T_v of 10% for stability and a period size of 2^{20} clock cycles. Finally, we use a decision interval of 32 periods (i.e. 1 observation period and 31 enforcement periods) and the Log-Decomposed MHA configuration search algorithm. These parameters were selected based on extensive simulations.

C.7.1 MHABC Performance

C.7.1.1 Performance Metric Results

We evaluate system performance with four metrics: *Harmonic Mean of Speedups (HMoS)*, *Fairness (F)*, *Aggregate Weighted Speedup (AWS)* and *Aggregate IPC (AI)*. In addition, we use the same metrics as online *policy metrics* which guide the

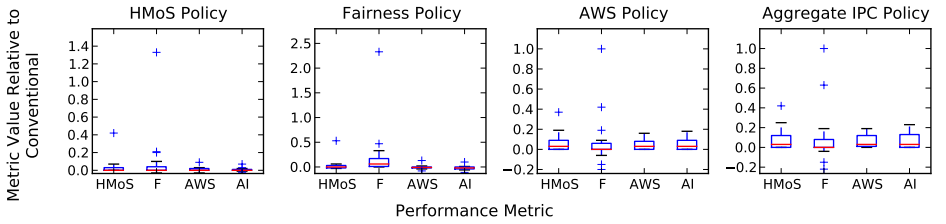


Figure C.9: MHABC 8-core Performance

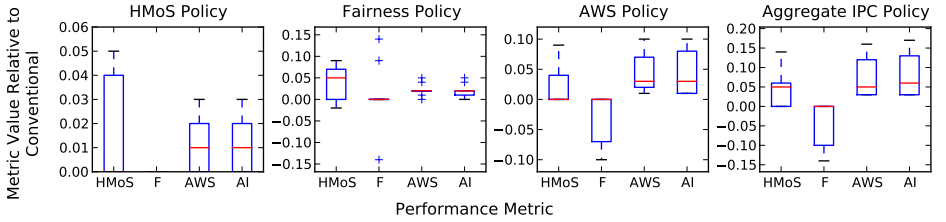


Figure C.10: MHABC 16-core performance

dynamic miss bandwidth allocations (recall Figure C.2 on page 135). We refer to MHABC with a particular policy metric as MHABC-HMoS, MHABC-F, etc. The use of AI as a performance metric has been criticized, and we only include it in the performance evaluations for completeness.

Figure C.8 shows MHABC's performance for the 4-core CMP with the four different policy metrics. Redistributing miss bandwidth can have a significant impact on the fairness-aware metrics HMoS and Fairness. For instance, MHABC-HMoS can improve HMoS by 106 % and Fairness by 200 %. This improvement does not hurt system throughput as the worst case AWS reduction for MHABC-HMoS is 3%.

Throughput and fairness can be conflicting objectives, and this is illustrated by the MHABC-AWS and MHABC-AI results. In some cases, MHABC-AWS is able to improve all performance metrics while in other cases it improves throughput and reduces fairness. MHABC-AI outperforms MHABC-AWS because it does not take private mode performance into account. Consequently, it aggressively reduces the bandwidth allocation of the low-IPC processes. If the memory system is severely congested, these processes spend most of their time waiting for memory. Consequently, the performance improvement from reduced congestion can outweigh the performance cost of reduced miss bandwidth. However, MHABC-AI is less stable than MHABC-AWS as its worst observed Fairness reduction is 42%. The worst Fairness reduction for MHABC-AWS is 26%.

Figure C.9 reports the performance of MHABC on our 8-core CMP. Again, MHABC-HMoS and MHABC-F provide substantial improvements on the HMoS and Fairness metrics with a low throughput cost. In addition, the performance impact of

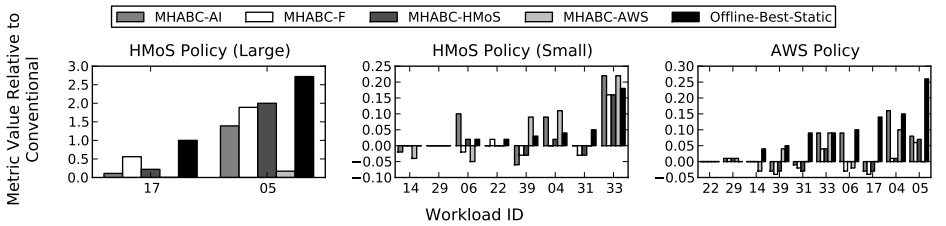


Figure C.11: MHABC Performance Compared to Offline-Best-Static

the throughput policies is larger than for the 4-core CMP. There are also fewer regressions on the fairness-aware metrics with the throughput policies. In fact, MHABC-AWS and MHABC-AI improve throughput without a single HMoS regression.

Figure C.10 shows the results for the 16-core CMP. Compared to the two other architectures, the improvements of MHABC are modest. However, they are consistent and all workloads experience AWS improvements with MHABC-AWS and MHABC-AI. The main reason for the modest improvements is that we scale the last-level cache size with the number of cores. Consequently, MHABC's potential is smaller for the 16-core model since improved cache sharing caused the large performance improvements for the other CMPs. Furthermore, managing cache space with bandwidth allocations becomes less effective as more processes share the on chip cache. Another observation in Figure C.10 is the unstable Fairness results for MHABC-Fairness. The reason is that this policy only considers the largest and smallest speedups which makes it unreliable with many processes.

C.7.1.2 Performance Relative to *Offline-Best-Static*

To approximate the ideal performance improvements from miss bandwidth allocations, we devised the *offline-best-static* configuration [14]. Offline-best-static is the best performing static asymmetric MHA chosen from a large collection of different MHAs. In this work, we try all combinations of 1, 2, 4 and 16 MSHRs in the last-level private cache on our 4-core CMP (i.e. $4^4 = 256$ MHAs). The large number of static asymmetric MHAs put a limit on the number of workloads and simulation points that can be simulated. Consequently, we limit our investigation to simulation point 0 and the 10 workloads with the highest effective bandwidth utilization for the conventional memory system.

Figure C.11 shows the HMoS and AWS results for MHABC and offline-best-static. For HMoS, we split the results into two plots to improve readability. MHABC can provide a significant improvement on the HMoS metric and does in some cases outperform offline-best-static due to its dynamic nature. For instance, MHABC-AWS and MHABC-AI provides better HMoS than offline-best-static for workload 33. On average, MHABC-HMoS is within 8% of offline-best-static for HMoS. MHABC-AI

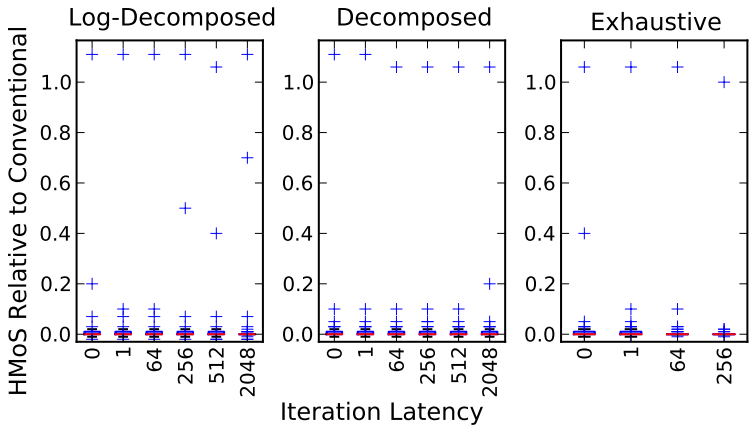


Figure C.12: MHA Configuration Search Algorithms

and MHABC-AWS are within 5% and 6% of offline-best-static for AWS, respectively.

MHABC does not achieve the ideal performance improvement for workload 17 which consists of *mgrid*, *facerec*, *mcf* and *swim*. Here, MHABC-F is able to improve HMoS by 56% while MHABC-HMoS provides an improvement of 22%. Furthermore, none of the policies are able to improve AWS. In this workload, *mcf* suffers severely from cache interference with *swim*. This interference can be alleviated if *swim*'s bandwidth allocation is reduced significantly. MHABC detects this interference, but cannot react until *mcf* tries to access the data *swim* replaced. This problem does not affect offline-best-static since the best static MHA is chosen when system performance is known. Consequently, MHABC's less than ideal performance on workload 17 is due its *reactive* nature.

C.7.1.3 Search Algorithms and Estimation Latencies

Figure C.12 shows the HMoS impact of different MHA configuration search algorithms and iteration latencies. For the exhaustive search policy, an iteration latency of 512 clock cycles would result in the allocation decision being available after the end of the period. Therefore, the largest iteration latency is 256 clock cycles for this algorithm. The main observation from Figure C.12 is that HMoS performance is similar for all algorithms and latencies. The search algorithms choose slightly different MHAs which result in different performance for the workloads where MHABC works well. Furthermore, the iteration latency influences the measurements which may result in MHABC selecting a different MHA. For the Log-Decomposed algorithm, the difference between two neighboring MHAs can be large. Therefore, it has a larger performance variation than the other algorithms.

Table C.6: Private Mode Estimate Relative Error

	Cores	Mean Error	Standard Deviation
Memory Latency	4	-1.1 %	10.1 %
	8	-2.1 %	8.8 %
	16	-0.8 %	4.8 %
IPC	4	-0.3 %	12.0 %
	8	-3.5 %	13.0 %
	16	-5.8 %	13.7 %

C.7.2 Performance Estimation Accuracy

Table C.6 shows the mean relative error and standard deviation of the private mode shared latency and performance estimates used in this work. Overall, these estimates are very accurate. To compare the shared mode estimates to the private mode values, it is important to obtain synchronized measurements from the two modes. For the shared latency estimates, we achieve this by reporting estimates and measurements every 1024 memory requests. For the performance measurements, we provide new estimates and measurements every 500000 committed instructions. We refer the reader to Jahre et al. for further experimental analysis of DIEF [15].

C.8 Discussion

In this work, we investigate a hardware implementation of the MHABC allocation policy. However, Nesbit et al. [27] argue that one should strive to implement allocation policies in software for flexibility. Policy flexibility is important for CMPs since they are used in a variety of settings. The minimum software interface of MHABC enables choosing different system performance metrics. To achieve more flexibility, we can make the performance measurements and DMHA mechanism available to the operating system and let the it choose the bandwidth allocations. In this way, MHABC can combine the ease of adoption of hardware techniques with the flexibility of software policies.

C.9 Related Work

The recent interest in resource allocation techniques for CMPs has resulted in attempts to use memory system measurements to predict performance change. Zhou et al. [42] proposed a model that predicts private mode performance from shared cache measurements. This model divides processor cycles into private cycles C_{pri}, cycles that are vulnerable to interference C_{vul} and overlap cycles C_{ovl}. Here, C_{ovl} are the cycles where both private and vulnerable work are carried out. Estimating C_{ovl} at runtime requires coordinated measurements from the processor core, private cache and shared cache. Our model is simpler to implement as C_{ovl} is captured

by M_p which is computed using the shared mode stall time for memory and the shared mode average memory latency. In addition, the OPACU methodology [24] provide dynamic estimates of the performance impact of cache capacity allocations. Finally, Srikantaiah et al. [39] showed how formal control theory can be applied to the cache capacity allocation problem. These proposals differ from this work in that they do not address how their models can be integrated with memory bus interference measurements.

There has been some research targeting resource allocation systems that cover multiple shared memory system units. Iyer et al. [13] proposed a high-level framework for implementing a QoS-aware memory system, while Nesbit et al. [27] proposed the Virtual Private Machines framework where a private virtual machine is created by dividing the available physical resources among applications. Srikantaiah and Kandemir [37] used shared cache capacity management to avoid memory bus congestion. Kaseridis et al. [20] proposed a resource management technique for multi-CMP systems that dynamically migrates processes to avoid congestion. In addition, Bitirgen et al. [3] showed how machine learning can be applied to the resource allocation problem. Finally, a number of researchers have looked the resource allocation problem for the shared cache [4, 12, 21, 29, 30, 38–41] and memory bus [11, 25, 26, 28, 33].

C.10 Conclusion

In this work, we show that shared memory system resource sharing can be improved by controlling miss bandwidth. By adjusting the maximum number of concurrent memory request for each process, we can alter sharing patterns and improve performance. We provide a practical bandwidth allocation system called *Miss Handling Architecture Bandwidth Control (MHABC)*. MHABC leverages the *Dynamic Miss Handling Architecture (DMHA)* enforcement mechanism, the *Dynamic Interference Estimation Framework (DIEF)* feedback mechanism and a novel allocation policy. DIEF provides estimates of private mode IPC with an average error of -0.3% and a standard deviation of 12.0%. MHABC's allocation policy models the performance effects of miss bandwidth allocations to choose an MHA that improves system performance. Consequently, MHABC-HMoS can improve HMoS by up to 106% at a worst-case throughput cost of 3%. Although we use our estimation methodology for miss bandwidth allocations in this work, the method is general and can be applied as long as the latency and MLP cost of a resource allocation can be quantified.

Acknowledgment

The authors thank Jan Christian Meyer for enlightening discussions of the ideas presented in this paper. In addition, we thank the Norwegian Metacenter for Com-

putational Science (NOTUR) for providing compute resources for our experiments. Lasse Natvig is a member of HiPEAC2 NoE.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California at Berkeley, 2009.

[2] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[3] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[4] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[5] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches. In *MEDEA '06: Proc. of the 2006 workshop on MEmory performance*, pages 45–52, 2006.

[6] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[7] K. I. Farkas and N. P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *ISCA '94: Proc. of the 21st An. Int. Symp. on Comp. Arch.*, pages 211–222, 1994.

[8] R. Gabor, S. Weiss, and A. Mendelson. Fairness and Throughput in Switch on Event Multithreading. In *MICRO 39: Proc. of the 39th Int. Symp. on Microarchitecture*, pages 149–160, 2006.

[9] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and More Flexible Program Analysis. In *Journal of Instruction Level Parallelism*, 2005.

[11] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA '08: Proc. of the 35th Int. Symp. on Computer Architecture*, pages 39–50, 2008.

[12] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th An. Int. Conf. on Supercomputing*, pages 257–266, 2004.

[13] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07*, pages 25–36, 2007.

[14] M. Jahre and L. Natvig. A High Performance Adaptive Miss Handling Architecture for Chip Multiprocessors. *Transactions on High Performance Embedded Architecture and Compilation*, 4(1), 2009.

[15] M. Jahre and L. Natvig. A Light-Weight Fairness Mechanism for Chip Multiprocessor Memory Systems. In *CF '09: Proc. of the 6th ACM Conf. on Computing Frontiers*, pages 1–10, 2009.

[16] M. Jahre, M. Grannæs, and L. Natvig. A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures. In *11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 622–629, 2009.

[17] M. Jahre, M. Grannæs, and L. Natvig. DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 292–306, 2010.

[18] *DDR2 SDRAM Specification*. JEDEC Solid State Tech. Association, May 2006.

[19] T. S. Karkhanis and J. E. Smith. A First-Order Superscalar Processor Model. *ISCA '04: Proceedings of the 31st An. Int. Symp. on Computer Architecture*, 2004.

[20] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John. A Bandwidth-aware Memory-subsystem Resource Management using Non-invasive Resource Profilers for Large CMP Systems. In *HPCA '10: Proc. of the 16th Int. Symp. on High-Performance Comp. Arch.*, 2010.

[21] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[22] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA '81: Proc. of the 8th An. Symp. on Comp. Arch.*, pages 81–87, 1981.

[23] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *ISPASS*, 2001.

[24] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Online Prediction of Applications Cache Utility. In *Int. Conf. on Embedded Comp. Systems: Architectures, Modeling and Simulation (IC-SAMOS)*, pages 169–177, 2007.

[25] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[26] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[27] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[28] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[29] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[30] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, pages 423–432, 2006.

[31] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *ISCA '06: Int. Symp. on Comp. Arch.*, pages 167–178, 2006.

[32] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[33] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[34] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[35] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Arch. Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[36] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[37] S. Srikantaiah and M. Kandemir. SRP: Symbiotic Resource Partitioning of the Memory Hierarchy in CMPs. In *Int. Conf. on High-Performance Embedded Architectures and Compilers*, 2010.

[38] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In *ASPLOS XIII: Proc. of the 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 135–144, 2008.

[39] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors. In *MICRO-42: Proc. of the Int. Symp. on Microarchitecture*, pages 517–528, 2009.

[40] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *ISCA '09: Proc. of the 36th annual Int. Symp on Computer Architecture*, pages 174–183, 2009.

[41] Y. Xie and G. H. Loh. Scalable Shared-Cache Management by Containing Thrashing Workloads. In *Int. Conf. on High-Performance Embedded Architectures and Compilers*, 2010.

[42] X. Zhou, W. Chen, and W. Zheng. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In *PACT '09: Proc. of the 18th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 384–393, 2009.

Appendix D

Paper B.I

A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures

Magnus Jahre, Marius Grannæs and Lasse Natvig
11th IEEE International Conference on High Performance Computing and Communications (HPCC)
2009

Abstract

The potential for destructive interference between running processes is increased as Chip Multiprocessors (CMPs) share more on-chip resources. We believe that understanding the nature of memory system interference is vital to achieve good fairness/complexity/performance trade-offs in CMPs. Our goal in this work is to quantify the latency penalties due to interference in all hardware-controlled, shared units (i.e. the on-chip interconnect, shared cache and memory bus). To achieve this, we simulate a wide variety of realistic CMP architectures. In particular, we vary the number of cores, interconnect topology, shared cache size and off-chip memory bandwidth. We observe that interference in the off-chip memory bus accounts for between 63% and 87% of the total interference impact while the impact of cache capacity interference can be lower than indicated by previous studies (between 5% and 32% of the total impact). In addition, as much as 11% of the total impact can be due to uncontrolled allocation of shared cache Miss Status Holding Registers (MSHRs).

D.1 Introduction

Chip Multiprocessors (CMPs) or multi-core architectures are the prevalent architecture for modern general-purpose, high-performance processors. In these architectures, it is common to share some part of the hardware-controlled memory system between cores. When multiple processes are run concurrently, the presence of shared resources makes destructive interference possible. In addition, the on-chip shared resources are managed by simple hardware policies that are unaware that the requests belong to different processes. The performance effects caused by destructive interference are hard to predict since they are a consequence of the runtime interaction between the memory request streams from co-scheduled processes. Consequently, destructive interference is an undesirable property and a considerable research effort has been aimed at developing techniques that reduce its performance impact [3, 17].

Figure D.1 illustrates that the current CMP memory systems are unable to provide predictable performance. To evaluate interference, we use a baseline configuration called the *private mode* where the benchmark is run in one of the processing cores while the remaining cores are idle. Consequently, it has exclusive access to all shared resources. Conversely, all benchmarks in a workload are run concurrently and compete for access to the shared resources in the *shared mode*. Figure D.1 shows the private- and shared mode IPCs of all benchmarks in two of our 40 randomly generated workloads. These measurements are taken from the 4-core, crossbar-based architecture with 4 memory channels which is the architectural configuration with the lowest amount of interference of the configurations used in this work. In workload 17, *facerec* and *mgrid* are heavily impacted by interference with a performance reduction of 46% and 21%, respectively. However, the performance of *mcf* is only reduced by 1%. This illustrates that the performance impact of interference can be substantial and that it does not affect all running processes equally. Furthermore, the performance impact of interference is unpredictable since *facerec* is only slowed down by 7% in workload 13. Since these effects are clearly undesirable, there is a need for architectural techniques that provide predictable performance and improve fairness.

Previously, cache capacity interference has received a great deal of attention [3, 6, 9, 12, 15, 21] while only a few researchers have proposed techniques that reduce memory bus interference [16, 17, 19]. Furthermore, there has been little interest in the details of designing a complete, thread-aware memory system [2, 10, 18]. A first step towards a unified approach to reducing interference in the hardware-managed memory system is to develop an understanding of the problem. For instance, we found that memory bus interference accounts for 64% of the total amount of interference while cache capacity interference only accounts for 25% with a powerful 4-channel memory bus in our 4-core crossbar-based CMP. When the complexity of current fair cache sharing techniques is taken into account, the fairness requirements on the system must be strict for thread-aware cache techniques to be worth the cost.

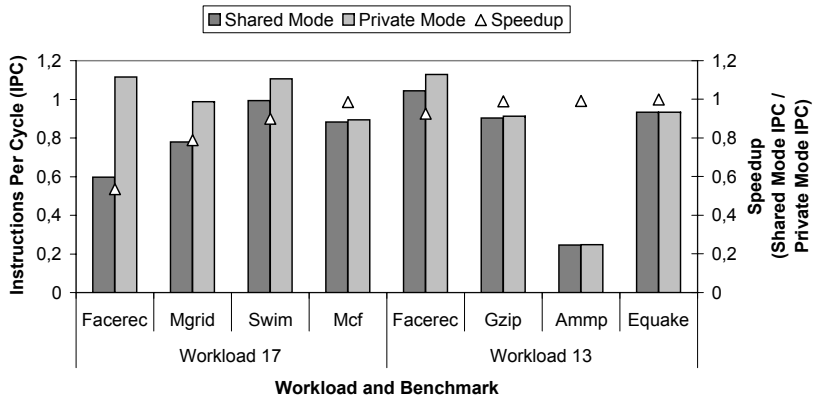


Figure D.1: Performance Impact of Interference in the 4-core, Crossbar-Based CMP with 4 Memory Channels

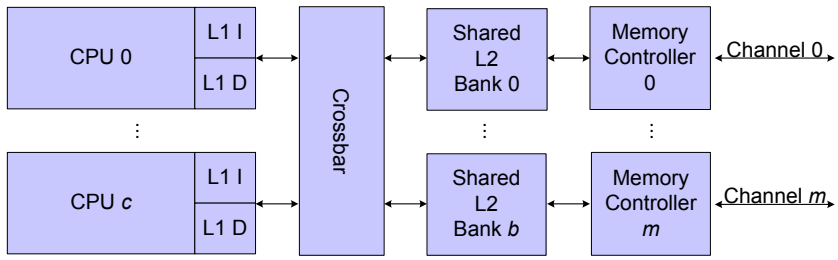


Figure D.2: Crossbar-based CMP

In this work, we aim to increase the understanding of the interference problem and thus help architects achieve good complexity/fairness trade-offs. This understanding is developed through detailed analysis of interference at the memory request level. Consequently, we are able to analyze both the relative interference impact of the different shared units as well as the distribution of interference penalties. Handling memory bus interference yields the largest gain, and we believe that employing a fairness-aware technique here will be sufficient for many architectures and usage scenarios. However, we have also observed interference due to shared cache Miss Status Holding Register (MSHR) allocation which must be handled if the fairness requirements are sufficiently strict. Finally, we show that the main driver of memory system interference is insufficient memory bus bandwidth. Since this parameter is limited by the number of physical pins on a chip and the electronic characteristics of the circuit board, it is likely that thread-aware memory bus schedulers will become a necessity in the near future.

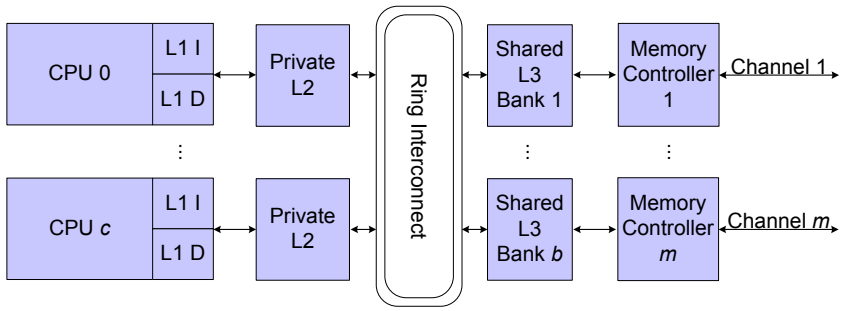


Figure D.3: Ring-based CMP

D.2 Related Work

It is common to aim an interference reduction technique at providing fairness and/or Quality of Service (QoS). A memory system is fair if the performance reduction due to interference between threads is distributed across all processes in proportion to their priorities [12]. QoS is provided if it is possible to put a limit on the maximum slowdown a process can experience when it is co-scheduled with any other process [3]. Furthermore, the allowed slow-down can depend on the priority of the process. In other words, the objective of fairness techniques is not to remove interference completely but to equalize its impact on all running processes.

There has been a considerable amount of research on how the performance impact from interference can be reduced in the hardware-controlled, shared memory system. However, most of these studies have focused on a single component of the entire system. For example, techniques have been proposed to reduce cache capacity interference [3, 6, 9, 12, 15, 21], cache bandwidth interference [20] and memory bus transfer interference [16, 17, 19]. Unfortunately, a technique that reduces interference in one component is not adequate to provide interference control for the complete memory system. Consequently, a few researchers have investigated how a chip-wide resource management technique can be designed. Iyer et al. [10] proposed a high-level framework for implementing a QoS-aware memory system, while Nesbit et al. [18] proposed the Virtual Private Machines framework where a private virtual machine is created by dividing the available physical resources among applications. In addition, Bitirgen et al. [2] showed how machine learning can be applied to the resource allocation problem. The focus of these works has been to partition all shared resources amongst processes according to some allocation policy. In this work, we investigate the impact of interference and provide guidance on how trade-offs can be handled in resource allocation implementations.

D.3 Methodology

D.3.1 Chip Multiprocessor Architectures

There is still considerable debate regarding the high-level organization of CMPs [7, 13, 23]. Therefore, we use two different CMP architectures that are similar to current general-purpose, high-performance CMP implementations for our interference investigations. Furthermore, we scale these architectures according to the expected improvements in process technology [8]. The first CMP type uses a crossbar interconnect to connect the private L1 caches to a large, shared L2 cache as shown in Figure D.2. Unfortunately, the crossbar does not scale in terms of area [14]. Consequently, we also use a different CMP model where a bi-directional ring is used as the interconnect. Since the ring has lower bandwidth than the crossbar, we add a private L2 cache to each processor to reduce the number of accesses to the interconnect. This is reasonable since the ring uses considerably less area than the crossbar. Furthermore, the number of processing cores and memory bus channels can be configured in both processor models which makes it possible to investigate the impact of memory system interference across a wide range of realistic CMP architectures. For convenience, we will refer to these architectures by the tuple *c-i-m* where c is the number of cores, i is the interconnect and m is the number of memory bus channels.

D.3.2 Measuring and Reporting Interference

To gather accurate interference measurements, it is convenient to compare to a baseline where interference does not occur [4]. In this work, we create such a baseline by letting the process run in one processing core and leaving the remaining cores idle. Consequently, the process has exclusive access to all shared resources and we will refer to this configuration as the *private mode*. Conversely, all processing cores are active and the processes compete for the shared resources in the *shared mode*. Mutlu and Moscibroda observed that memory system interference is related to the memory latencies in the shared and private modes with the formula: $\textit{interference penalty} = \textit{shared mode latency} - \textit{private mode latency}$ [16].

In our CMP models, there are three shared units: the interconnect, the memory bus and the shared cache. To assess the interference impact of each of these units, we partition the memory request latency through the shared memory system as shown in Table D.1. For the interconnect, we divide the latency into three types: entry, transfer and delivery. The interconnect has a finite entry queue. If this queue becomes full, the interconnect can not accept any more requests and the request is delayed in the private cache MSHR. We refer to this as *Interconnect Entry Interference* if it causes a different delay in the shared mode than in the private mode. Furthermore, the shared cache can block. In this case, all requests waiting behind a request for a blocked bank are delayed since reordering requests can cause star-

Table D.1: Shared Memory System Latency Breakdown

Type	**Description**
Interconnect Entry	The number of cycles a request was kept in the private cache MSHR before it is accepted into a interconnect queue
Interconnect Transfer	The number of cycles spent in the interconnect queue plus the interconnect transfer latency
Interconnect Delivery	The number of cycles a request was delayed because a shared cache bank could not accept requests due to insufficient buffer space
Memory Bus Entry	The number of cycles a request was delayed in a shared cache MSHR before it was accepted into a memory controller queue
Memory Bus Transfer	The number of cycles a request spent in the memory controller queue plus the number of cycles used to retrieve the requested data from DRAM
Cache Capacity	The number of cycles used to service misses that would not occur if the process had exclusive access to the shared cache

vation. We refer to interference arising from this situation as *Interconnect Delivery Interference*. Finally, *Interconnect Transfer Interference* is the difference between the shared mode and private mode latencies when there is no cache blocking.

In the memory bus, we divide the latency into two types: entry and transfer. Again, the entry delay is the number of cycles the request is kept in an MSHR before it is accepted into the memory bus queue. If this latency is different for the shared and private modes, we refer to it as *Memory Bus Entry Interference*. In addition, *Memory Bus Transfer Interference* is the difference between the memory bus queue latency plus service latency in the two modes. Since there is no buffer allocation in the shared cache on a response, the memory bus does not have a delivery latency.

Finally, competition for space in the shared cache can lead to *Cache Capacity Interference*. Unlike the interference types discussed above, cache capacity interference does not have a latency value directly associated with it. The key observation is that if a request experiences a bus transfer latency in the shared mode and no bus transfer latency in the private mode, we have a miss in the shared cache that would have been a hit if the process had the entire cache to itself. The extra latency caused by this event in our CMP models is the number of cycles used to service the request in the memory bus. Consequently, the latency penalty of cache capacity interference is the sum of the bus entry latency and the bus transfer latency of the request.

Figure D.4 illustrates the two stage process of gathering interference measurements and aggregating them for a single architecture. In the first stage, we create a compact representation of the measured interference for each benchmark in all work-

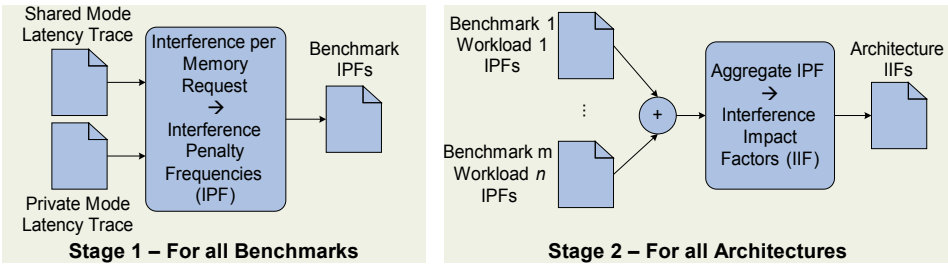


Figure D.4: Interference Measurement Workflow

loads and architectures. First, we record the latency of all shared mode memory requests and all private mode memory requests. For all shared mode requests, we find the corresponding private mode request and compute the interference penalties for all interference types. If there are more than one request for the same address, we assume that the requests occur in the same order in both the private and shared modes. Then, we create a histogram representation of the data by counting the number of requests that experience a certain interference penalty for each interference type. For example, if a request for memory address 15 experiences 12 cycles of interconnect transfer interference, we add 1 to the interconnect transfer interference entry at position 12. We refer to this data as the *Interference Penalty Frequency (IPF)*, and stage 1 of the analysis is complete when we have created IPF files for all workloads and architectures.

Stage 2 is the process of aggregating the per benchmark IPF files into one file for each architecture. First, we sum the request counts for each interference penalty from all files belonging to the architecture of interest. For some of the interference types, it is very common to not experience interference. These entries are of little interest and will dominate the results if we use plot the number of requests per interference penalty directly. Consequently, we devise a new metric called the *Interference Impact Factor (IIF)* that balances the latency penalty of interference against the probability of it arising (i.e. $IIF(i) = i \cdot P(i)$). For example, an experiment that results in 15 requests with 3 cycles interconnect transfer interference and 100 requests in total gives $IIF(3) = 3 \cdot \frac{15}{100}$. When we have computed the IIFs for all interference penalties, stage 2 is finished. In most cases, there is a large range of possible interference values and there is a need to summarize the IIFs for a range of interference penalties into a single number. To do this, we use the *Aggregate Interference Impact Factor (AIIF)* which is simply the sum of the IIFs for all or a subset of the observed interference penalties (i.e. $AIIF(a, b) = \sum_{i=a}^{b} IIF(i)$).

D.3.3 Processor Model Scaling

To investigate the impact of interference in multi-core architectures, it is important that reasonable parameters are used to scale the latency, bandwidth and capacity

Table D.2: Architecture Parameter Scaling

	Crossbar Based Architecture			**Ring Based Architecture**		
	4-core	**8-core**	**16-core**	**4-core**	**8-core**	**16-core**
ITRS Year of Production	2007	2010	2013	2007	2010	2013
Feature Size (nm)	65	45	32	65	45	32
Shared Cache Size (MB)	8	16	32	8	16	32
Memory Bus Channels	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4	1, 2 or 4
Interconnect Latency (End-to-End/Per Hop)	8/-	16/-	30/-	-/4	-/4	-/8

Table D.3: Cache Parameters (4-core/8-core/16-core)

	L1 Private Cache	**L2 Private Cache**	**L2/L3 Shared Cache**
Size	64KB	1 MB	8/16/32 MB
Associativity	2	4	16
Access Latency (cycles)	3/2/2	9/6/5	16/12/12
Cycle Time (cycles)	2	4/3/2	4
MSHRs / WB (per bank)	16MSHRs/4WB	16	16/32/64
Banks	1	1	4
Area (mm^2)	2.3/1.1/0.5	14.6/7.0/3.6	94.0/91.9/84.7

of the various units in the memory system. To this end, we have used the International Technology Roadmap for Semiconductors [8] to estimate scaling trends and CACTI 5.3 [25] to find reasonable caches for the multi-core architectures used in this work. Table D.2 summarizes the main multi-core model parameters. With each improvement in feature size, we double the number of processing cores but use the same core implementation. Furthermore, we follow the ITRS expectation that the interconnect transfer latency will roughly double with each technology generation. The only exception is the per hop latency of the 4-core ring architecture which we assume is limited by the cache cycle time. To account for this latency increase, we double the ring bandwidth across generations. Since the ITRS projections for off-chip bandwidth results in a large range of possible pin counts, we simulate all architectures with 1, 2 and 4 independent memory channels.

Table D.3 contains the parameters of our scaled on-chip caches. Here, we choose to keep the percentage of the total chip area occupied by L2 and L3 caches in the ring-based CMP constant. We use the same shared cache for the crossbar based CMP, but here we only use two levels of caches. Consequently, we assume that the area made available by using a two level cache hierarchy is sufficient to implement a crossbar interconnect. To reduce the shared cache access time and increase the opportunity for cache access parallelism, we divide the shared cache into 4 banks.

Table D.4: Processor Core Parameters

Parameter	Value
Clock frequency	4 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	4 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 4-way 2048 entry BTB

Table D.5: Interconnect and DRAM Interface

Parameter	Value
Crossbar Interconnect	8/16/30 cycles end-to-end transfer latency, 32 entry request queue, Pipelined (2/4/6 pipe stages)
Ring Interconnect	4/4/8 cycles per hop transfer latency, 1/1/2 pipe stages per hop, 32 entry request queue, 1/2/2 request rings, 1 response ring
Point to Point Link	4/3/2 transfer latency, 32 entry request queue
Main memory	DDR2-800, 4-4-4-12 timing, 64 entry read queue, 64 entry write queue, 1 KB pages, 8 banks, FR-FCFS scheduling [22], Closed page policy

D.3.4 Simulation Methodology

We use the system call emulation mode of the cycle-accurate M5 simulator [1] for our experiments. The processor architecture parameters for the simulated CMPs are shown in Table D.4. Table D.5 contains the interconnect and memory bus parameters, and the cache parameters are outlined in Table D.3. We have extended M5 with crossbar and ring interconnects and a detailed DDR2-800 memory bus and DRAM model [11]. For the shared mode, we generated 40 different 4-core workloads (Table D.6), 20 8-core workloads (Table D.7) and 10 16-core workloads (Table D.8) by picking benchmarks at random from the full SPEC CPU2000 benchmark suite [24]. The only requirement given to the random selection process is that a benchmark can only appear once in each workload. These workloads are fast-forwarded for 1 billion clock cycles before we gather traces for 100 million clock cycles. For our interference measurement methodology to be accurate, it is critical to minimize the difference between the memory requests in the shared and private modes. To ensure this, we use static cache partitioning and an infinite bandwidth interconnect and memory bus during fast forwarding such that the simulation sample starts on a similar instruction in both modes. Furthermore, we run the shared mode experiments first and then retrieve the number of instructions the benchmark committed. Then, we run the private mode simulation for the exact same number of instructions.

Since our processor cores are out-of-order, we can get cache misses from wrong path instructions that only occur in either the private or shared mode. Secondly, the start and termination of the simulation sample is not perfectly synchronized

Table D.6: Randomly Generated 4-core Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	mesa, twolf, art, vpr	9	crafty, twolf, bzip, perlbmk	17	mgrid, facerec, mcf, swim	25	twolf, crafty, bzip, art	33	swim, gap, vortex1, perlbmk
2	art, vortex1, applu, crafty	10	eon, twolf, galgel, crafty	18	equake, applu, eon, gzip	26	applu, gap, perlbmk, crafty	34	equake, twolf, bzip, galgel
3	gap, eon, art, wupwise	11	vortex1, eon, art, equake	19	galgel, mesa, gzip, gcc	27	galgel, facerec, eon, mesa	35	applu, eon, fma3d, vortex1
4	fma3d, applu, parser, swim	12	gzip, lucas, twolf, apsi	20	art, galgel, parser, eon	28	vpr, crafty, applu, vortex1	36	lucas, ammp, twolf, fma3d
5	mcf, swim, gzip, vortex1	13	facerec, ammp, gzip, equake	21	bzip, gzip, perlbmk, eon	29	twolf, vpr, swim, wupwise	37	eon, parser, bzip, mcf
6	swim, galgel, apsi, applu	14	swim, sixtrack, mgrid, vortex1	22	vpr, swim, apsi, gcc	30	parser, mesa, vortex1, gcc	38	vpr, vortex1, wupwise, applu
7	gzip, wupwise, eon, equake	15	sixtrack, fma3d, parser, mcf	23	art, applu, perlbmk, mesa	31	lucas, mgrid, sixtrack, gap	39	lucas, mgrid, swim, gzip
8	sixtrack, gcc, facerec, perlbmk	16	twolf, galgel, crafty, applu	24	facerec, eon, bzip, mesa	32	facerec, galgel, vpr, sixtrack	40	gzip, swim, eon, fma3d

between the two modes. Thirdly, our memory controller reorders requests to achieve high page hit rates which can affect the private cache access patterns and miss rates. For these reasons, there can be small differences between the private and shared mode memory request traces. We remove these differences by applying two preprocessing steps before analyzing the traces. Firstly, we remove the requests for addresses that only occur in the private or shared modes. Secondly, we remove the superfluous requests of the mode that has the most requests in the cases where there are a different number of requests for the same address in the shared and private modes. These steps result in the removal of 0.1% of the observed requests.

D.4 Results

Modern out-of-order processors and memory systems contain a substantial amount of logic dedicated to hiding memory latency. Since our interference measurement methodology is latency focused, it is necessary to verify that the observed inter-

Table D.7: Randomly Generated 8-core Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	ammp, mcf, vpr, fma3d, equake, sixtrack, galgel, bzip	6	applu, mcf, perlbmk, parser, crafty, eon, galgel, fma3d	11	ammp, lucas, wupwise, eon, twolf, fma3d, gcc, equake	16	apsi, ammp, vortex1, vpr, gap, perlbmk, art, bzip
2	crafty, vortex1, facerec, ammp, bzip, parser, mcf, perlbmk	7	fma3d, gzip, lucas, perlbmk, bzip, apsi, crafty, gap	12	mcf, galgel, gap, gzip, swim, sixtrack, vpr, fma3d	17	gzip, art, equake, facerec, eon, apsi, gcc, wupwise
3	lucas, vpr, mesa, apsi, swim, art, gzip, twolf	8	swim, gzip, ammp, facerec, perlbmk, equake, gcc, apsi	13	mesa, fma3d, gap, lucas, wupwise, galgel, sixtrack, parser	18	perlbmk, gap, parser, swim, sixtrack, fma3d, lucas, vortex1
4	art, mcf, perlbmk, wupwise, ammp, applu, mesa, swim	9	gap, mcf, vpr, apsi, vortex1, lucas, parser, applu	14	bzip, mgrid, facerec, art, eon, swim, equake, apsi	19	lucas, mesa, apsi, fma3d, mcf, parser, crafty, gcc
5	eon, apsi, equake, vpr, fma3d, facerec, gcc, vortex1	10	mcf, sixtrack, vpr, swim, gzip, mgrid, ammp, lucas	15	swim, vpr, gap, facerec, twolf, sixtrack, mcf, crafty	20	gcc, perlbmk, sixtrack, parser, vortex1, eon, facerec, galgel

ference result in an asymmetric performance reduction. To this end, we use the fairness metric of Gabor et al. [5]. This metric expresses the difference between the largest and smallest shared mode slowdowns for one workload and provides values in the range from 0 to 1 where 1 indicates that the slowdown is the same for all benchmarks. A value of 0 indicates that at least one benchmark is not making forward progress.

Figure D.5 shows the distribution of fairness metric values for all 4-core CMPs used in this work. Here, we plot the lowest fairness value observed when a certain number of workloads are taken into account for the different CMP architectures. The main observation from Figure D.5 is that many workloads have reasonably good fairness values. However, there are also workloads where interference leads to large performance differences between the benchmarks (i.e. low fairness). This supports the claim that interference-aware techniques are necessary to reduce performance variability.

Figure D.6 shows the interference results for all architectures examined in this work. The main observation is that memory bus transfer interference is the major interference contributor across all architectures. This trend is also visible in Figure D.5. Cache capacity interference is the second most important source of interference, but its impact is considerably smaller than the impact of bus interference. In addition, there are architectures (e.g. 16-CB-4) where the impact of cache capacity interference is small. Finally, there is more interconnect transfer interference in the crossbar interconnect than in the ring for the 16-core CMP. This seemingly counter

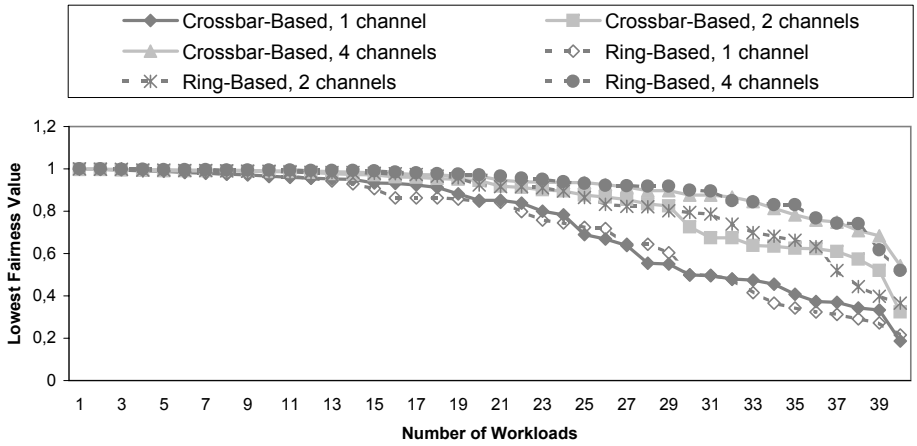


Figure D.5: 4-core Fairness Metric Values

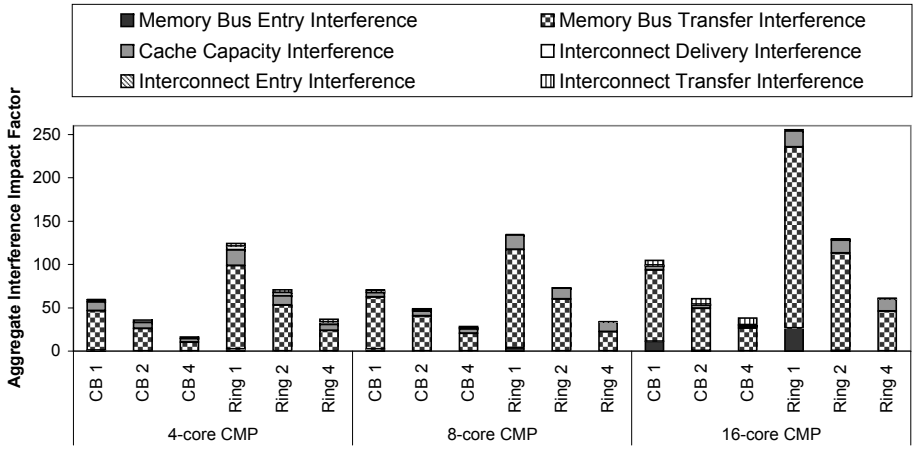


Figure D.6: Interference Impact Breakdown

Table D.8: Randomly Generated 16-core Multiprogrammed Workloads

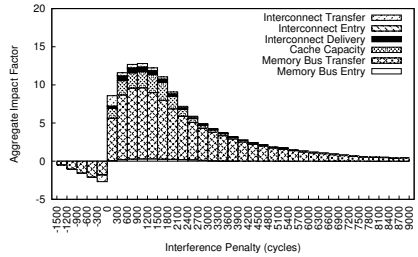
ID	Benchmarks	ID	Benchmarks
1	lucas, art, ammp, bzip, sixtrack, vpr, gzip, fma3d, equake, gcc, vortex1, facerec, galgel, crafty, apsi, twolf	6	parser, mesa, bzip, vortex1, vpr, fma3d, gap, gcc, perlbmk, gzip, mcf, crafty, eon, equake, facerec, galgel
2	lucas, ammp, mgrid, bzip, swim, crafty, galgel, equake, vortex1, parser, vpr, eon, wupwise, gzip, twolf, mcf	7	gzip, sixtrack, gap, fma3d, eon, galgel, perlbmk, art, bzip, ammp, equake, lucas, parser, facerec, apsi, crafty
3	lucas, ammp, art, bzip, twolf, applu, facerec, apsi, mesa, eon, swim, galgel, gzip, crafty, gap, perlbmk	8	perlbmk, gzip, apsi, twolf, wupwise, gap, vpr, mgrid, galgel, facerec, gcc, eon, mcf, lucas, fma3d, ammp
4	crafty, twolf, mgrid, applu, wupwise, swim, parser, fma3d, mesa, perlbmk, facerec, gcc, lucas, vortex1, galgel, bzip	9	mgrid, art, facerec, gcc, vpr, gzip, parser, ammp, fma3d, galgel, crafty, applu, twolf, bzip, mcf, apsi
5	bzip, facerec, vortex1, ammp, gzip, swim, fma3d, equake, lucas, apsi, applu, vpr, perlbmk, sixtrack, mcf, mesa	10	apsi, swim, crafty, art, sixtrack, ammp, galgel, lucas, vortex1, gzip, perlbmk, vpr, gcc, mesa, gap, equake

intuitive result is due to two factors. Firstly, the ring-based architecture has a private L2 cache that reduces the pressure on the interconnect. Secondly, we do not increase the number of banks in the shared cache which reduces the parallelism available in the crossbar.

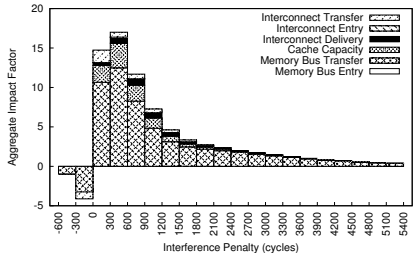
Figure D.7 shows the interference distribution for the 4-core CMP for both interconnects and all memory bus configurations used in this work. Here, the interference impact factors are aggregated into bins of size 300, and we remove all bins that have a AIIF value of less than 0.35 to improve readability. While Figure D.6 showed that interference is reduced when more memory bus bandwidth is made available, Figure D.7 illustrates that the interference distribution also changes significantly. For the bandwidth constrained architectures (e.g. Figure D.7(a) and D.7(d)), the interference impact increases to a maximum before it decreases. In the 4-channel architectures (Figure D.7(c) and D.7(f)), the largest interference impact is in the 0 to 300 bin and the impact decreases rapidly. The interference impact of the low penalty bins is significantly higher for the 4-channel architectures but the total impact is lower because of the distribution's short tail.

Figure D.7 illustrates that the cache capacity interference impact is heavily dependent on the amount of memory bus interference. The reason is that the cost of cache capacity interference is the memory bus service time of the additional requests. Furthermore, the impact from interconnect transfer interference is small across all architectures. Although this interference type occurs very frequently, the interference penalty is small which results in a low interference impact. In addition, there is some interconnect delivery interference in all architectures which is due to shared cache blocking. The impact from this type of interference is large enough that it most likely must be dealt with in architectures with strict QoS requirements.

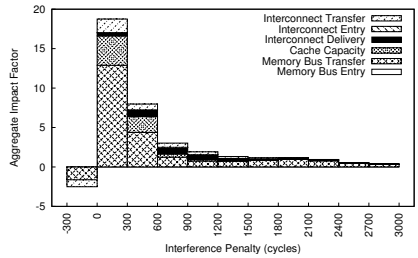
There is also a considerable amount of constructive interference. With the 4-Ring-1 architecture (Figure D.7(a)), constructive memory bus interference leads to a noticeable impact in the -1500 to -1200 cycles bin. This can be explained by taking



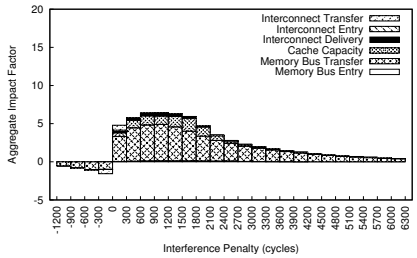
(a) 4-Ring-1



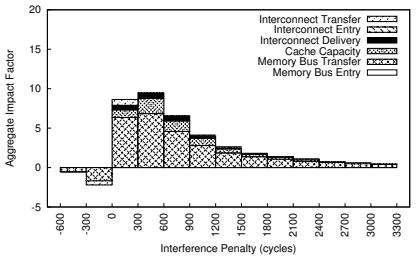
(b) 4-Ring-2



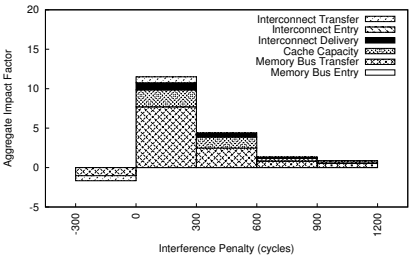
(c) 4-Ring-4



(d) 4-Crossbar-1

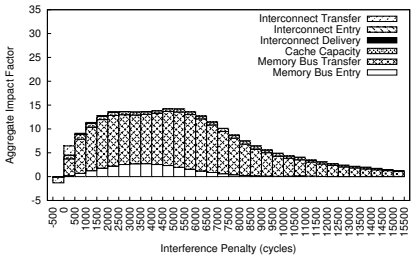


(e) 4-Crossbar-2

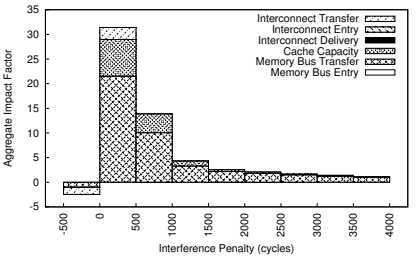


(f) 4-Crossbar-4

Figure D.7: 4-core CMP Interference Impact (*cores-interconnect-channels*)



(a) 16-Ring-1



(b) 16-Ring-4

Figure D.8: 16-core Ring Interference Impact

into account that our memory controller allows some requests to skip past the queue to achieve higher page hit rates and better memory bus utilization [22]. For the interconnect transfer interference, the impact from constructive interference is much lower. In this case, the constructive interference is due to some benchmarks having significant interconnect delays when they have the memory bus to themselves. In the shared mode, memory bus interference reduces execution speed enough that the interconnect congestion disappears which results in lower transfer delays in the shared mode.

To illustrate the impact on interference by increasing the number of processing cores, we show the results of two 16-core ring-based architectures in Figure D.8. Here, we use a bin size of 500 and only show bins that have an AIIF value of 1.0 or more. As expected, Figure D.8(a) shows that there is a large amount of interference if the memory bus bandwidth is not scaled with the number of cores. Furthermore, memory bus entry interference has a considerable impact for this architecture. Consequently, a significant part of the interference is due to shared cache misses not being accepted into the memory bus queue because it is full. This further illustrates the need for fair buffer management observed in all 4-core architectures. Figure D.8(b) shows the effect of increasing the number of memory bus channels to 4. Here, the distribution has a considerably shorter tail. However, the impact of the 0 to 500 cycle bin is large which indicates that low-penalty interference is frequent. In other words, providing more resources reduces the impact of interference but does not remove it. This indicates that fairness techniques are useful even when there are no severe performance bottlenecks.

D.5 Conclusion and Further Work

In this work, we have shown that the impact of interference will increase as more cores are added to the chip by investigating a variety of realistic CMP architectures with 4, 8 and 16 cores. Consequently, techniques that reduce this interference are needed in future CMPs. We found that memory bus interference is the major source of interference and it is responsible for between 63% and 87% of the total interference impact depending on the architectures. Furthermore, it is unlikely that this situation will improve in the future as memory bus bandwidth is limited by the number of physical pins on a chip and the electronic characteristics of the circuit board. We also observed that cache capacity interference can be a relatively small part of the total interference impact (between 5% and 32%). Consequently, adding a fair memory controller might be sufficient to achieve acceptable fairness and QoS for many near-term architectures. However, we have also observed architectures where 11% of the total interference impact is due to the shared cache MSHR allocation policy for which no solutions are currently known.

In this work, we have developed an understanding of memory system interference that can be useful for future research. However, we have only investigated CMPs where no fairness techniques have been implemented. A possible avenue of further

work is to investigate how implementing fairness techniques in one shared unit will influence the interference impact of the other shared units. For instance, a cache capacity sharing technique might reduce the overall number of cache misses enough to reduce the impact of memory bus interference. On the other hand, it can potentially increase the number misses by limiting the cache space available to a process which might result in more memory bus interference. In addition, we observed that shared cache blocking and memory controller blocking can be important contributors to interference in certain architectures. One possible solution to this problem is to allocate MSHR entries and memory bus queue space per thread. However, this must be done carefully to ensure that the provided resources are utilized efficiently.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[3] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[4] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[5] R. Gabor, S. Weiss, and A. Mendelson. Fairness and Throughput in Switch on Event Multithreading. In *MICRO 39: Proc. of the 39th Int. Symp. on Microarchitecture*, pages 149–160, 2006.

[6] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *MICRO 40: Proc. of the 40th An. IEEE/ACM Int. Symp. on Microarchitecture*, 2007.

[7] H. Hofstee. Power Efficient Processor Architecture and the Cell Processor. *HPCA 11: 11th Int. Symp. on High-Performance Comp. Arch.*, pages 258–262, 2005.

[8] ITRS. International Technology Roadmap for Semiconductors - 2007 Edition. `http://www.itrs.net/`, 2007.

[9] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS '04: Proceedings of the 18th An. Int. Conf. on Supercomputing*, pages 257–266, 2004.

[10] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07*, pages 25–36, 2007.

[11] *DDR2 SDRAM Specification*. JEDEC Solid State Tech. Association, May 2006.

[12] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[13] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[14] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *ISCA '05: Proc. of the 32nd Int. Symp. on Comp. Arch.*, pages 408–419, 2005.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *HPCA '08: Proc. of the 13th Int. Symp. on High-Perf. Comp. Arch.*, 2008.

[16] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[17] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[18] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[19] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[20] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[21] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural Support for Operating System-driven CMP Cache Management. In *PACT '06: Proc. of the 15th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 2–12, 2006.

[22] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[23] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008*, pages 1–15, 2008.

[24] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[25] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical report, HP Laboratories Palo Alto, 2008.

Appendix E

Paper B.II

DIEF: An Accurate Interference Feedback Mechanism for Chip Multiprocessor Memory Systems

Magnus Jahre, Marius Grannæs and Lasse Natvig

5th International Conference on High Performance and Embedded Architectures and Compilers
2010

Abstract

Chip Multi-Processors (CMPs) commonly share hardware-controlled on-chip units that are unaware that memory requests are issued by independent processors. Consequently, the resources a process receives will vary depending on the behavior of the processes it is co-scheduled with. Resource allocation techniques can avoid this problem if they are provided with an accurate interference estimate. Our Dynamic Interference Estimation Framework (DIEF) provides this service by dynamically estimating the latency a process would experience with exclusive access to all hardware-controlled, shared resources. Since the total interference latency is the sum of the interference latency in each shared unit, the system designer can choose estimation techniques to achieve the desired accuracy/complexity trade-off. In this work, we provide high-accuracy estimation techniques for the on-chip interconnect, shared cache and memory bus. This DIEF implementation has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

E.1 Introduction

Chip Multi-Processors (CMPs) commonly share parts of the memory system. While some CMPs have private caches and only share off-chip bandwidth, other CMPs share an on-chip interconnect and cache space between cores. This resource sharing is often beneficial since it can improve resource utilization compared to a private design and facilitates efficient inter-core communication. However, sharing may also adversely affect performance when the system resources are insufficient for co-scheduled processes. This is due to the use of rudimentary hardware policies like First Come First Served (FCFS) and Least Recently Used (LRU) which were primarily designed for use in single-core processors. These policies do not provide predictable resource allocations because processes with higher access frequencies receive a larger part of the shared resource [10, 15]. Since CMPs often run multiprogrammed workloads, the performance of a single process can be heavily influenced by the processes it is co-scheduled with.

Resource allocation techniques that attempt to alleviate interference problems, commonly aim their effort at improving *fairness* and/or *Quality of Service (QoS)*. A memory system is fair if the performance reduction due to interference between threads is distributed across all processes in proportion to their priorities [9]. QoS is provided if it is possible to put a limit on the maximum slowdown a process can experience when co-scheduled with any other process [3]. Nesbit et al. [12] propose a high-level architecture for resource allocation systems which divide the system into three independent, cooperating modules. Here, the *feedback mechanisms* provide measurements of the current resource utilization and/or the performance of the running programs. Then, the *allocation policy* decides on a new and improved resource allocation and implements this with the *allocation mechanisms*. Since resource allocations do not change very often, allocation policies should be implemented in software to achieve flexibility. On the other hand, allocation and feedback mechanisms that interact closely with the hardware resources, must be implemented in hardware for efficiency.

In this work, we provide the first detailed implementation of a unified feedback mechanism for the hardware-managed, shared memory system called the Dynamic Interference Estimation Framework (DIEF). DIEF dynamically estimates the average memory latency a process would experience if it had exclusive access to all shared resources. In addition, DIEF measures the actual shared memory latency to establish the relative latency impact from sharing effects. Choosing average memory latency as the interference metric has the advantage that the total interference latency is the sum of the interference latency of each shared unit. Consequently, the system designer can choose interference estimation techniques that achieve the appropriate accuracy/complexity trade-off. Since processing cores can hide latency, an allocation policy needs a performance-oriented feedback mechanism to complement DIEF which can be provided by well-known techniques like performance counters [19].

In this work, we aim our efforts at providing an accurate DIEF implementation. To accomplish this, we develop interference measurement mechanisms for ring and crossbar interconnects, shared caches and a multi-channel DDR2 memory bus. These mechanisms are tested on a variety of CMP architectures with 4, 8 or 16 cores, 2 or 3 cache levels and 1, 2 or 4 memory bus channels. DIEF is very accurate for these architectures and has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

E.2 Background

E.2.1 Interference Definition and Metrics

When evaluating CMP memory system fairness, it is convenient to compare to a baseline where interference does not occur. One way of creating such a baseline is to let the process run in one processing core of the CMP and leave the remaining cores idle [5, 11]. Consequently, the process has exclusive access to all shared resources, and we will refer to this configuration as the *private mode*. Conversely, all processing cores are active and the processes compete for shared resources in the *shared mode*. We refer to a baseline created in this way as a *Single Program Baseline (SPB)*.

It is also possible to create a fairness baseline by statically partitioning all shared resources equally among the processors [3]. We refer to this baseline type as a *Multiprogrammed Baseline (MPB)*. The main advantage of MPB is that it exists in the shared mode. Consequently, it is easy to ensure that a fairness technique does not perform worse than the baseline. However, MPB also has three major disadvantages. Firstly, it only accounts for interference in the resources that have been statically and equally partitioned. This can lead to erroneous results if important interference sources are missed. Secondly, static and equal division of DRAM bandwidth does not lead to a static and equal division of latency [10]. The reason is that the latency of a request depends heavily on which requests was issued before it. Consequently, it may be difficult to implement a good static and equal sharing baseline for the memory interface. Finally, the relationship between performance and resource allocation is rarely linear [15]. Consequently, a process may experience severe performance degradation in the statically shared baseline. If a fairness technique then removes this degradation, one might be lead to believe that the technique also improves throughput when the degradation in fact was due to the baseline's suboptimal resource allocation.

These problems can be avoided by using the Single Program Baseline (SPB). Unfortunately, SPB does not exist in the shared mode. By definition, it requires that the performance in the shared mode is compared to the interference-free private mode. In this work, we provide a feedback mechanism that estimates SPB latency at runtime. We define the interference I_i experienced by a request i as the difference between the shared mode latency L_i and private mode latency $\mathcal{L}_i$ (i.e.

$I_i = L_i - \mathcal{L}_i$). This definition is an extension of the interference definition by Mutlu and Moscibroda [10].

The shared mode estimate of the private mode latency $\hat{\mathcal{L}}_i$ may be different from the actual private mode latency $\mathcal{L}_i$. Consequently, it is important that a feedback mechanism minimizes the difference between these values. We define the measurement error for request i to be $E_i = \hat{\mathcal{L}}_i - \mathcal{L}_i$. Since the interference estimate $\hat{I}$ is related to the private mode latency estimate $\hat{\mathcal{L}}$ by the formula $\hat{\mathcal{L}}_i = L_i - \hat{I}_i$, the feedback mechanism can choose to estimate either $\hat{\mathcal{L}}_i$ or $\hat{I}_i$ and compute the other. A dynamic resource allocation technique will use $\hat{\mathcal{L}}$ to establish the relative impact of interference on the different running processes. Consequently, the impact of the error depends on the shared mode latency L. To account for this we define the relative error $\mathcal{E}_i = E_i / L_i$. We aggregate multiple errors by using the arithmetic mean, standard deviation and root mean squared error of E and $\mathcal{E}$.

E.2.2 Modern Memory Bus Interfaces

Memory bus scheduling is a challenging problem due to the 3D structure of DRAM consisting of rows, columns and banks. Commonly, a DRAM read transaction consists of first sending the row address, then the column address and finally receiving the data. When a row is accessed, its contents are stored in a register known as the row buffer, and a row is often referred to as a *page*. If the row has to be activated before it can be read, the access is referred to as a *row miss* or *page miss*. It is possible to carry out repeated column accesses to an open page, called *row hits* or *page hits*. This is a great advantage as the latency of a row hit is much lower than the latency of a row miss. The situation where two consecutive requests access the same bank but different rows is known as a *row conflict* and is very expensive in terms of latency. DRAM accesses are pipelined, so there are no idle cycles on the memory bus if the next column command is sent while the data transfer is in progress. Furthermore, command accesses to one bank can be overlapped with data transfers from a different bank.

Rixner et al. [17] proposed the First Ready - First Come First Served (FR-FCFS) algorithm for scheduling DRAM requests. Here, memory requests are reordered to achieve high page hit rates which result in increased memory bus utilization. This algorithm prioritizes requests according to three rules: prioritize ready commands over commands that are not ready, prioritize column commands over other commands and prioritize the oldest request over younger requests.

E.3 Shared Memory System Latency Taxonomy

The main advantage of measuring interference in terms of average round trip latency through the shared memory system is that the total interference of a single request is the sum of the interference it experiences in each of the shared units.

Table E.1: Memory System Latency Taxonomy

Module	Type	Description	SM	PM	Int.
Interconnect	Entry (ie)	The number of cycles a request is kept in the private cache MSHR before it is accepted into an interconnect queue	L_i^{ie}	$\mathcal{L}_i^{\mathrm{ie}}$	I_i^{ie}
	Queue (iq)	The number of cycles spent in the interconnect queue	L_i^{iq}	$\mathcal{L}_i^{\mathrm{iq}}$	I_i^{iq}
	Transfer (it)	The number of cycles spent on transferring the request from source to destination	L_i^{it}	$\mathcal{L}_i^{\mathrm{it}}$	I_i^{it}
	Delivery (id)	The number of cycles a request was delayed because a shared cache bank could not accept requests due to insufficient buffer space	L_i^{id}	$\mathcal{L}_i^{\mathrm{id}}$	I_i^{id}
Shared Cache	Capacity (cc)	The number of cycles used to service a miss that would not occur if the process had exclusive access to the shared cache	-	-	I_i^{cc}
Memory Controller	Entry (me)	The number of cycles a request was delayed in a shared cache MSHR before it was accepted into a memory controller queue	L_i^{me}	$\mathcal{L}_i^{\mathrm{me}}$	I_i^{me}
	Queue (mq)	The number of cycles a request spent in the memory controller queue	L_i^{mq}	$\mathcal{L}_i^{\mathrm{mq}}$	I_i^{mq}
	Transfer (mt)	The number of cycles the request occupied the memory data bus	L_i^{mt}	$\mathcal{L}_i^{\mathrm{mt}}$	I_i^{mt}
Shared Memory System	Total	The total number of cycles a request uses through the entire hardware-controlled, shared memory system	L_i	$\mathcal{L}_i$	I_i

Consequently, it is possible to independently implement and validate the feedback mechanism for each source of interference. In this work, we develop a comprehensive view of memory system interference which is shown in Table E.1.

The hardware-controlled, shared memory system commonly consists of three types of units. Firstly, an interconnect is needed to connect the private caches to one or more shared caches. Secondly, there can be one or more levels of shared caches with varying sharing degrees. Finally, off-chip bandwidth can be shared between cores. Although the organization of these shared units will vary from CMP to CMP, we believe that this model captures the essential types of interference in the hardware-controlled, shared memory system.

Within these units, the shared resources are either *bandwidth* or *capacity*. In the memory bus and interconnects, bandwidth is the main shared resource. However, memory requests are kept in finite buffers while waiting for access to the shared transmission channels. Consequently, there are also different forms of capacity interference in these units. We divide the latency through the units where bandwidth

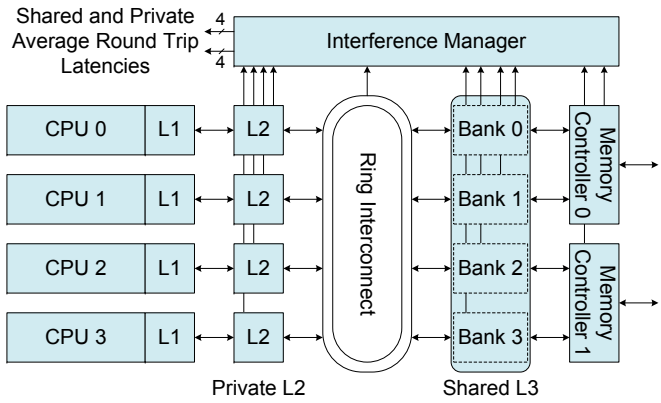


Figure E.1: Dynamic Interference Estimation Framework (DIEF) Architecture

is the main shared resource into four parts. The *entry* latency is the latency the request experiences while waiting to be accepted into the input queue. Then, the *queue* latency is the number of cycles it spends in the queue before it is granted access to the resource. The next latency type is the *transfer* latency which is the number of cycles it takes to transfer the request from source to destination. Finally, it might not be possible to deliver the request if the destination lacks sufficient buffer space. In this case, the request experiences an additional *delivery* latency. There is no delivery latency in the memory bus since the last level cache must be able to receive responses to avoid deadlocks.

To provide system-wide, latency-based interference measurements, the latency cost of shared cache interference misses must be established. This problem can be solved by observing that interference misses are associated with the latency penalty of retrieving the data from the next cache level or memory. If we assume one level of shared caches, the cache capacity interference experienced by request i is the sum of request i's memory bus entry, queue and transfer latency ($I_i^{\mathrm{cc}} = L_i^{\mathrm{me}} + L_i^{\mathrm{mq}} + L_i^{\mathrm{mt}}$). For convenience, we use the first letter of the shared unit (i.e. i, c or m) and the first letter of the latency type (i.e. e, q, t, d or c) to produce a two-letter identifier (e.g. **i**nterconnect **e**ntry is **ie**).

E.4 The Dynamic Interference Estimation Framework

The purpose of a dynamic interference estimation technique is to provide a reliable measure of how memory system interference affects the running processes. In this work, we propose the Dynamic Interference Estimation Framework (DIEF)

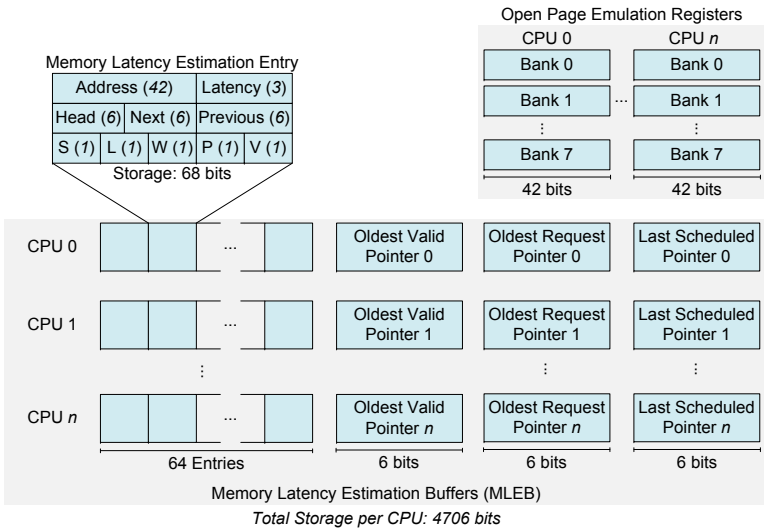


Figure E.2: Private Memory Bus Emulation

that continuously monitors all shared units to provide accurate interference estimates. Figure E.1 shows DIEF's high-level architecture where each shared unit is augmented with extra functionality (not on the unit's critical path) that measures interference and/or latencies at runtime. These measurements are continuously communicated to the Interference Manager which uses it to measure the shared mode average round trip latency L and create an estimate $\hat{\mathcal{L}}$ of the private mode latency $\mathcal{L}$. Since memory bus interference is the interference type with the largest impact, most of our efforts are directed at estimating this latency type [7]. The operating system must inform DIEF of context switches to ensure that the measurements are not polluted by the actions of other processes. In the case of multi-threaded applications, the operating system also needs to instruct DIEF to treat the application's set of processing cores as one entity. Without loss of generality, we consider the situation where each core runs one single-threaded application in the remainder of this work.

E.4.1 Estimating Private Memory Bus Latency ($\hat{\mathcal{L}}^{\mathrm{mt}}$, $\hat{\mathcal{L}}^{\mathrm{mq}}$ and $\hat{\mathcal{L}}^{\mathrm{me}}$)

E.4.1.1 Estimating Transfer and Queue Latencies ($\hat{\mathcal{L}}^{\mathrm{mt}}$ and $\hat{\mathcal{L}}^{\mathrm{mq}}$)

Modern memory bus scheduling algorithms reorder requests to improve memory bus throughput [17]. Therefore, the *execution order* of memory requests depend on the memory bus queue contents and can be very different in the shared and private

Table E.2: Status Bits

S	Transfer latency estimation $\hat{\mathcal{L}}^{\mathrm{mt}}$ is valid
L	$\hat{\mathcal{L}}^{\mathrm{mq}}$ and $\hat{\mathcal{L}}^{\mathrm{mt}}$ has been computed
W	The request is a write
P	Entry is private mode only
V	Entry is valid

Table E.3: $\hat{\mathcal{L}}^{\mathrm{mt}}$ Estimates

Prev. State	**Next State** Read Bank i	Write Bank i
Hit (any bank)	40	40
Miss (any bank)	120	110
Conflict Read Bank i	200	190
Conflict Write Bank i	260	250
Conflict Read Bank j	170	160
Conflict Write Bank j	260	250

modes. However, the *arrival order* of requests is very similar. Consequently, it is possible to estimate the private mode *execution order* by emulating the private scheduling algorithm on the shared mode requests. Then, the private *execution order* and bank state determine the transfer latency estimate $\hat{\mathcal{L}}^{\mathrm{mt}}$. The queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}$ can be estimated by following the private *execution order* and accumulating transfer latencies.

Figure E.2 shows the hardware support needed to emulate a private memory bus. This hardware is not on the critical path and consists of n *Memory Latency Estimation Buffers (MLEB)* (one for each processor). Each time the memory controller receives a request from a certain CPU, it is added to the corresponding MLEB. When the request is serviced by the memory controller, the state stored in this buffer is used to estimate its private mode queue latency $\hat{\mathcal{L}}^{\mathrm{mq}}$ and transfer latency $\hat{\mathcal{L}}^{\mathrm{mt}}$. This calculation can be allowed to take on the order of tens of processor cycles since the memory bus is commonly clocked at a much lower frequency than the processing core.

Each estimation entry has a head pointer, a next pointer and a previous pointer. The previous/next pointers store the private *execution order* by pointing to the element that was scheduled before/after the request in the private mode. The head pointer points to the estimation entry that was the next to be serviced when the request was added, and it is used to estimate queue latency. Furthermore, each entry contains five status bits: S, L, W, P and V. These are explained in Table E.2. Finally, the *Oldest Valid Pointer* points to the oldest valid MLEB entry, the *Oldest Request Pointer* points to the oldest non-serviced entry and the *Last Scheduled Pointer* points to the most recently scheduled entry.

To improve estimation accuracy, we add the Open Page Emulation Registers. These where originally proposed by Mutlu and Moscibroda [10] and are used to estimate whether a request is a page hit, miss or conflict. Here, each register holds the address of the last accessed memory page. These registers are also used to schedule requests according to the FR-FCFS scheduling algorithm [17].

Generally, there are more queued requests in the MLEB than in the private mode memory bus queue since competition for the bus is more severe in the shared

Algorithm 7 Private Memory Bus Queue and Transfer Latency Estimation

procedure ESTIMATEPRIVATELATENCIES(Memory request r)
  while r not serviced **do**
    Emulate FR-FCFS scheduling of elements within horizon given by the Page Locality Factor
  end while
  Initialize request pointer c to point to head(r) and queue latency $\hat{\mathcal{L}}_{r}^{mq}$ to 0
  while c is not equal to r **and** c is scheduled before r **do**
    Increment queue latency $\hat{\mathcal{L}}_{r}^{mq}$ with the transfer latency $\hat{\mathcal{L}}_{c}^{mt}$ of request c
    Update c by following the next pointer of c
  end while
  Invalidate any entries that are no longer needed to compute queue and transfer latencies
  return transfer latency $\hat{\mathcal{L}}_{r}^{mt}$ **and** queue latency $\hat{\mathcal{L}}_{r}^{mq}$ of request r
end procedure

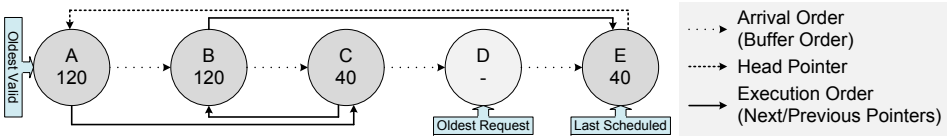


Figure E.3: Memory Bus Queue and Transfer Latency Estimation Example

mode. This can result in overestimating the number of page hits if the process has sufficient page locality. To account for this, we add a parameter called the *Page Locality Factor*. This factor determines the number of estimation entries that should be examined while looking for a page hit. Setting the page locality factor to 1 assumes no reordering in the private memory system.

If we ignore the effects of shared cache interference, the requests that reach the memory bus are the ones that are not filtered out by the on-chip caches. Since we use the same cache hierarchy in the shared configuration and the baseline, the order of the memory request are nearly identical but their timing will be different. However, there may be differences resulting from the interleaving of writebacks and reads since the memory controller may reorder requests differently in the two configurations. When cache interference is taken into account, the request stream can be very different. Consequently, the shared cache interference technique should identify both private- and shared-only requests and communicate this information to the memory bus interference technique.

Finally, we need to produce estimates of the shared mode queue latency. This can be accomplished by adding a register for all queue entries and incrementing it with the memory bus transfer latency every time a request is finished. Alternatively, a request can be assigned a timestamp on arrival and this timestamp can then be compared to the value of a counter when the request is issued.

The Latency Estimation Algorithm Algorithm 7 summarizes the estimation algorithm for the private memory bus transfer latency $\hat{\mathcal{L}}^{mt}$ and queue latency $\hat{\mathcal{L}}^{mq}$. We illustrate the estimation procedure with the example in Figure E.3. There are five queued requests, and request E has just been serviced by the shared mode

memory controller. To determine the transfer latency $\hat{\mathcal{L}}_{\mathrm{E}}^{\mathrm{mt}}$ of E, the estimation algorithm emulates scheduling requests within the limit given by the Page Locality Factor. In this example, request A is serviced first and its transfer latency is estimated. Then, request C is serviced before B since it is a private mode page hit. Finally, request E is serviced before D since it accesses the same page as B which gives $\hat{\mathcal{L}}_{\mathrm{E}}^{\mathrm{mt}} = 40$. Then, we can estimate $\hat{\mathcal{L}}_{\mathrm{E}}^{\mathrm{mq}}$ by following E's head pointer to A and accumulating the transfer latencies of all elements between A and E in the private execution order. Consequently, the queue latency estimate for E is $\hat{\mathcal{L}}_{\mathrm{E}}^{\mathrm{mq}} = \hat{\mathcal{L}}_{\mathrm{A}}^{\mathrm{mt}} + \hat{\mathcal{L}}_{\mathrm{C}}^{\mathrm{mt}} + \hat{\mathcal{L}}_{\mathrm{B}}^{\mathrm{mt}} = 120 + 40 + 120$.

There are a number of possible transfer latencies due to different active pages, overlapping of commands with data transfers from other banks and timing constraints regarding when a bank can be precharged. However, we observed that only a small number of these possible latencies occur frequently in the private mode. Consequently, it is possible to store the most common transfer latencies in a lookup table. Then, the latency is determined by whether the previous and next requests are to the same bank and whether they are reads or writes. This lookup table is created at design time by analyzing the private mode access behavior for the chosen memory bus type. Table E.3 shows the lookup table of the DDR2 memory bus used in this work.

A private-only entry (P bit set) can be invalidated when its latency is not needed to compute the queue latency of any other element. For shared mode entries, the latency of the entry must also be computed before it can be deleted. In addition, we require that the most recently scheduled element is not invalidated. The deletion algorithm is based on the observation that the head pointer of the oldest undeletable element e in the *arrival order* will point to the oldest head element h in the *arrival order*. Consequently, we know that all elements after h in the *execution order* are needed to compute the queue latency for e. If an entry has been removed due to insufficient buffer space, we use the last computed transfer and queue latency.

E.4.1.2 Estimating Memory Bus Entry Interference $\hat{I}^{\mathrm{me}}$

When the memory bus queue becomes full, the memory controller blocks and the requests remain in the shared cache MSHRs. We account for this interference by observing that the maximum number of requests a processor core can issue simultaneously is the sum of MSHRs and writeback buffers in the last-level private cache. Furthermore, the shared buffers will be dimensioned to handle roughly c times this number of requests (c = number of cores) since too few buffers will lead to frequent performance bottlenecks. The effect of this observation is that a single core will not be able to fill the buffers in the shared part of the memory system. Consequently, any shared mode latency due to memory bus blocking is interference.

E.4.2 Estimating Cache Capacity Interference $\hat{I}^{\text{cc}}$

To identify shared cache interference misses, we use an Auxiliary Tag Directory (ATD) [4, 16] per core. Each time a request is received in the shared cache, the request is inserted into the ATD belonging to the processor that sent the request. Consequently, the ATD contains the tags the processor would have had in the shared cache if it was running alone. On each access, we compare the output from the ATD with the output from the actual cache. If the request is a hit in the ATD and a miss in the real cache, we store a timestamp and tag the request as a shared mode only cache miss. This bit is used to keep the request out of the memory bus private mode latency estimation. When the request has been serviced in the memory bus and returned to the cache, we retrieve its latency and communicate it to the Interference Manager as cache capacity interference. We also record if an ATD entry would have been written to in the private mode. In this case, a replacement would have triggered a writeback in the private mode. When this happens, we insert a private mode only writeback request into the memory bus private mode latency estimation.

In this work, our aim is to accurately measure interference. Consequently, we are willing to invest a fair bit of area into making the estimates accurate. We use CACTI version 5.3 [20] to establish that the size of each ATD is roughly 4% of the shared cache area. Qureshi et al. [16] showed that sampling as few as 16 to 32 sets can be sufficient to represent cache behavior. With 32 sets, the area of each ATD is reduced to around 0.01 % of the shared cache area. In DIEF, using set sampling is not straight forward since the memory bus interference estimation mechanism needs to know which misses are shared-only interference misses. This problem can likely be avoided at the cost of reduced accuracy by using an estimated interference miss probability to select requests for the memory bus interference estimation. The area overhead can be further reduced at the cost of accuracy and measurement latency by time multiplexing the ATDs. Work in this direction is underway.

E.4.3 Estimating Interconnect Interference ($\hat{I}^{\text{ie}}$, $\hat{I}^{\text{iq}}$, $\hat{I}^{\text{it}}$ and $\hat{I}^{\text{id}}$)

The main component of interconnect interference is due to requests having to wait for access to the shared transmission medium ($\hat{I}^{\text{iq}}$). It is easy to measure interference in the ring and crossbar interconnects used in this work since latency is independent of access order. If a processor i is not able to issue a request because a request r from processor j is being transferred, we add the number of cycles request r occupied the transmission medium for each delayed request from processor i to the interference estimate. Since the interconnects may be pipelined, the number of cycles a processor delays another processor may be less than the transfer latency. In the ring interconnect, the transfer latency depends on which core the process is scheduled on and this needs to be taken into account when estimating interference. Again, we assume that all blocking due to full buffers is interference.

Table E.4: CMP Models

Interconnect	#CPUs	Process	Private Cache	Shared Cache	Memory Bus
Crossbar, 8/16/30 cycles end-to-end transfer latency, 32 entry queue	4	65 nm	2-way 64KB L1 Data, 2-way 64KB L1 Inst.	16-way 8MB L2	DDR2-800, 4-4-4-12 timing, 8 banks, 1KB pages, 64 entry read queue, 64 entry write queue, FR-FCFS, Open Page Policy
	8	45 nm		16-way 16MB L2	
	16	32 nm		16-way 32MB L2	
Ring, 4/4/8 cycles per hop transfer latency, 32 entry queue	4	65 nm	2-way 64KB L1 Data, 2-way 64KB L1 Inst., 4-way 1MB Unified L2	16-way 8MB L3	
	8	45 nm		16-way 16MB L3	
	16	32 nm		16-way 32MB L3	

E.5 Methodology

We use the system call emulation mode of the cycle-accurate M5 simulator [1] for our experiments and have extended M5 with crossbar and ring interconnects as well as a detailed DDR2-800 memory bus and DRAM model [8]. We model two CMP architectures that are similar to current general-purpose, high-performance CMP implementations and identify these models by the name of the on-chip interconnect (i.e. crossbar or ring). Table E.4 summarizes the CMP models used in this work, and a further discussion of the models is provided by Jahre et al. [7]. The only difference between Jahre et al.'s configuration and ours is that we use an open page policy in the memory controller. We also use Jahre et al.'s 40 4-core workloads, 20 8-core workloads and 10 16-core workloads that were generated by picking benchmarks at random from the full SPEC CPU2000 benchmark suite [18]. The only requirement given to the random selection process is that a benchmark can only appear once in each workload. These workloads are fast-forwarded for 1 billion clock cycles before we run detailed simulation for 100 million clock cycles. To achieve synchronized measurements of L and $\mathcal{L}$, it is critical to minimize the difference between the memory requests in the shared and private modes. To ensure this, we use static cache partitioning and an infinite bandwidth interconnect and memory bus during fast forwarding such that the simulation sample starts on a similar instruction in both modes. Furthermore, we run the shared mode experiments first and then retrieve the number of instructions the benchmark committed. Then, we run the private mode simulation for the exact same number of instructions.

E.6 Results

In this section, we present the results from our experiments with DIEF. When not otherwise stated, we use our best performing configuration with 8192 requests per sample, a page locality factor of 3 and a 64 entry bus estimation buffer. These values were found empirically by extensive simulation.

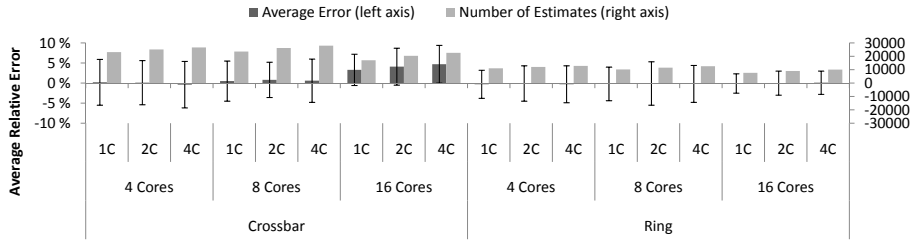


Figure E.4: Relative Estimation Errors and Number of Estimates

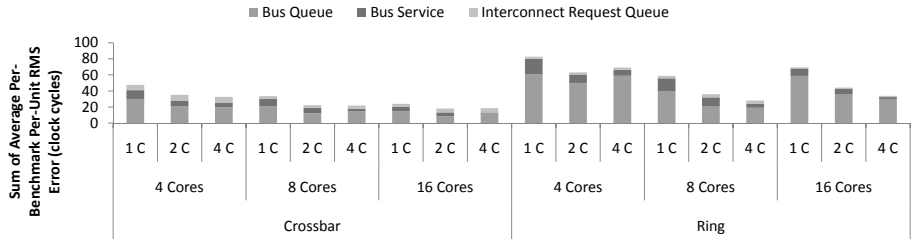


Figure E.5: Interference Estimation Error Breakdown

E.6.1 Estimation Accuracy

Figure E.4 shows the average relative error and one standard deviation of all estimates produced by DIEF. In addition, Figure E.4 contains the number of estimates used to compute these statistics. We use the abbreviations 1C, 2C and 4C to represent 1 memory bus channel, 2 memory bus channels and 4 memory bus channels, respectively. The main observation is that the average error is close to zero in all cases. Furthermore, the standard deviation is at most 5.8%.

Figure E.5 breaks down the average root mean squared (RMS) error for all architectures used in this work. We have removed all interference types where the average RMS error is less than 2 clock cycles to improve readability. Furthermore, cache capacity interference is not included since it has no corresponding private mode latency. Figure E.5 shows that most of the measurement error is due to the memory bus queue estimate $\hat{\mathcal{L}}^{\mathrm{mq}}$. This is not surprising as our queue latency estimation model does not cover the case where a request is delayed by page hits that arrive after it. Furthermore, our model does not accurately predict the difference between the number of simultaneously queued requests in the two models. However, given the good average accuracy shown in Figure E.4, the measurements are likely accurate enough to be used by a dynamic fairness technique. Another observation is that the absolute measurement error is larger in the ring architectures. This is due to poor utilization of the L3 cache because the private L2 caches reduce the access frequency. Consequently, a cache thrashing process is able to evict a larger

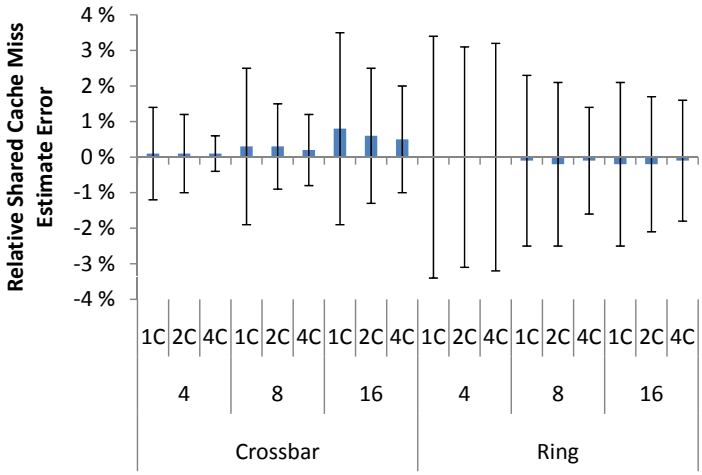


Figure E.6: ATD Estimation Error

amount of the data needed by a less cache intensive thread which in turn puts a larger strain on the memory bus.

To quantify the accuracy of the ATD interference miss estimates, we count the number of actual misses and the number of interference misses. Here, the shared mode miss count estimate is computed by subtracting the number of additional shared cache misses identified by the ATD from the shared mode miss count. Then, we adapt the relative error metric to cache misses by using the estimated number of misses $\hat{\mathcal{M}}$, the actual private mode number of misses $\mathcal{M}$ and the shared mode number of misses M ($\mathcal{E} = (\hat{\mathcal{M}} - \mathcal{M})/M$). Figure E.6 shows that our ATD-based interference miss estimation has an average relative estimation error of at most 0.8% and maximum standard deviation of 3.4%.

Figure E.7 shows the distribution of the memory bus queue RMS errors for the 4-core CMP models. Here, we represent the measurement error for each instance of a benchmark by the average RMS error of the estimates for this benchmark. Then, we sort the average RMS errors such that each point in the figure represents the maximum average RMS error observed for a certain percentage of benchmarks. Figure E.7 shows that the memory bus queue estimates are very accurate. When 60% of the benchmarks are taken into account, the worst average RMS error observed for any architecture is 20 clock cycles. However, there is a short tail where the measurement error is significant. Since the average round trip memory latency is high in these cases, the values are most likely good enough to be used by dynamic resource allocation techniques. Finally, the lines stop at 82% for the ring architecture and 97% for the crossbar architecture because some benchmarks have too few memory requests to produce any estimates.

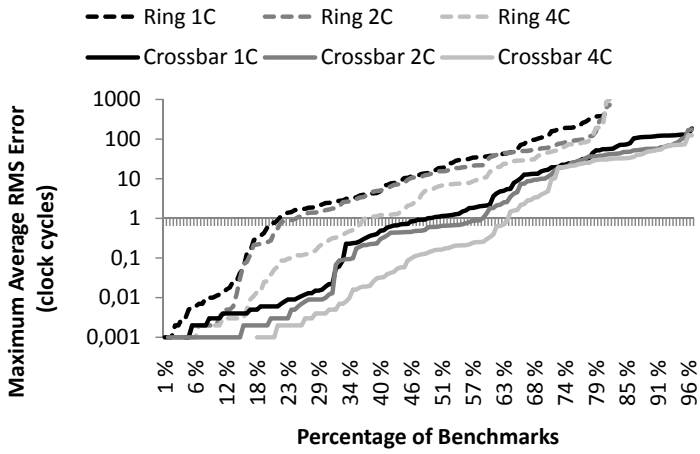


Figure E.7: 4-core Bus Queue Error

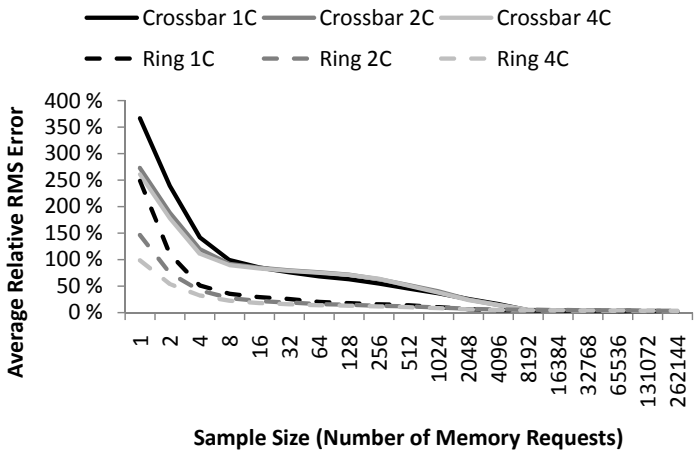
E.6.2 DIEF Parameters

In this section, we provide an empirical analysis of DIEF's main parameters: sample size, page locality factor and memory bus estimation buffer size. The choice of sample size is a trade-off between achieving low variability and receiving new estimates often enough to make high quality resource allocation decisions. Figure E.8 shows the average relative RMS error and average latency between estimates for the 8-core architectures. Our choice of 8192 requests per sample is on the flat part of the error plot and has an acceptable average latency.

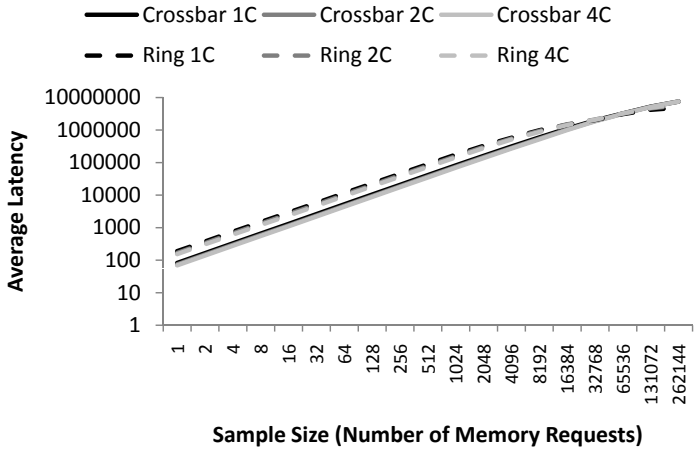
Figure E.9 shows the average RMS error for different page locality factors. The general trend is that the page locality factor should be low because there is usually more locality in the shared mode estimation buffer than in the private memory bus queues. This is because a larger number of requests are available to the scheduler in the shared mode due to more competition. A page locality factor of 3 is the best overall. Finally, Figure E.10 shows the error resulting from varying the memory bus estimation buffer size. Here, 64 entries are necessary to achieve low error for the ring architecture.

E.7 Related Work

A few researchers have addressed the issue of dynamic interference measurement in CMPs. Cache Scouts [21] is a shared cache interference measurement technique that estimates interference by counting the number of cache blocks that are evicted by different processors. Consequently, they assume that all blocks evicted by a



(a) Root Mean Squared Error



(b) Average Latency Between Estimates

Figure E.8: 8-core CMP Sample Size Accuracy Impact

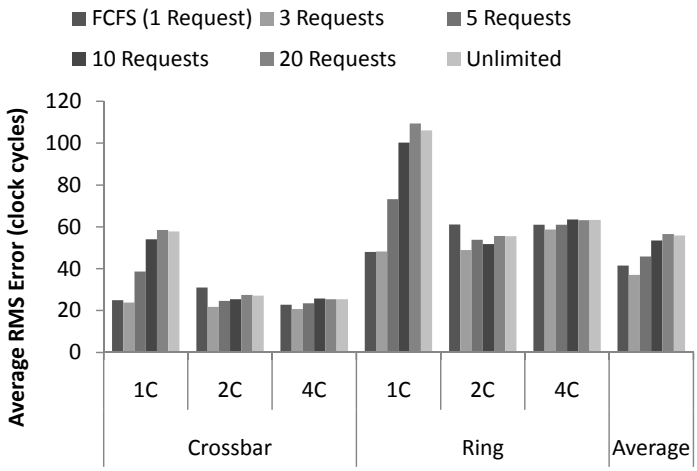


Figure E.9: 4-core Page Locality Factor

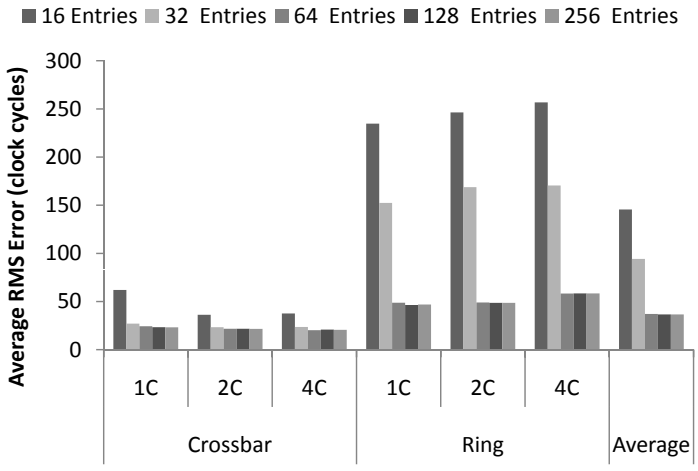


Figure E.10: 4-core Bus Buffer Size

different processor would have been reused which may lead to measurement errors. Mutlu and Moscibroda [10] propose a run-time interference measurement technique that they use to guide a memory bus scheduling algorithm in a system with private caches.

Most previous studies that aim to improve resource sharing in CMP memory systems, have focused on a single component of the entire system. For example, techniques have been proposed to reduce cache capacity interference (e.g. [3, 9]), cache bandwidth interference [14] and memory bus interference [10, 11, 13]. In addition, a few researchers have investigated how a chip-wide resource management technique can be designed. Iyer et al. [6] proposed a high-level framework for implementing a QoS-aware memory system, while Nesbit et al. [12] proposed the Virtual Private Machines framework where a private virtual machine is created by dividing the available physical resources among applications. In addition, Bitirgen et al. [2] showed how machine learning can be applied to the resource allocation problem.

E.8 Conclusion

Accurate feedback mechanisms are needed to implement robust resource allocation systems in future CMPs. In this work, we propose the Dynamic Interference Estimation Framework (DIEF) which is the first detailed implementation of a unified feedback mechanism for CMP memory systems. DIEF is a collection of techniques that cooperate to estimate the average memory latency a process would experience if it had exclusive access to all shared resources. Choosing the average memory latency as the unit of interference has the advantage that the total memory latency is the sum of the latency in each shared unit. Consequently, CMP designers can choose estimation techniques that achieve the desired accuracy/complexity trade-off for each shared unit. In this work, we describe a high accuracy DIEF implementation which has an average relative estimate error between -0.4% and 4.7% and a standard deviation between 2.4% and 5.8%.

Acknowledgments

This project was supported in part by the Norwegian Metacenter for Computational Science (NOTUR). Lasse Natvig is a member of HiPEAC2 NoE.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO 41: Proc. of the 41th IEEE/ACM Int. Symp. on Microarchitecture*, 2008.

[3] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS '07: Proc. of the 21st Annual Int. Conf. on Supercomputing*, pages 242–252, 2007.

[4] H. Dybdahl, P. Stenstrom, and L. Natvig. An LRU-based Replacement Algorithm Augmented with Frequency of Access in Shared Chip-Multiprocessor Caches. In *MEDEA '06: Proc. of the 2006 workshop on MEmory performance*, pages 45–52, 2006.

[5] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.

[6] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07*, pages 25–36, 2007.

[7] M. Jahre, M. Grannæs, and L. Natvig. A Quantitative Study of Memory System Interference in Chip Multiprocessor Architectures. In *11th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 622–629, 2009.

[8] *DDR2 SDRAM Specification.* JEDEC Solid State Tech. Association, May 2006.

[9] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT '04: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[10] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[11] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proc. of the 35th An. Int. Symp. on Comp. Arch.*, pages 63–74, 2008.

[12] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore Resource Management. *IEEE Micro*, 28(3):6–16, 2008.

[13] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[14] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proc. of the 34th An. Int. Symp. on Comp. Arch.*, pages 57–68, 2007.

[15] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proc. of the 39th An. IEEE/ACM Int. Symp. on Microarch.*, pages 423–432, 2006.

[16] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *ISCA '06: Int. Symp. on Comp. Arch.*, pages 167–178, 2006.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[18] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[19] B. Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 22 (4):64–71, 2002.

[20] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical report, HP Laboratories Palo Alto, 2008.

[21] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Arch. and Comp. Tech.*, pages 339–352, 2007.

Appendix F

Paper C.I

Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors

Marius Grannæs, Magnus Jahre and Lasse Natvig
XXVI IEEE International Conference on Computer Design (ICCD)
2008

Abstract

The pressure on off-chip memory increases significantly as more cores compete for the same resources. A CMP deals with the memory wall by exploiting thread level parallelism (TLP), shifting the focus from reducing overall memory latency to memory throughput. This extends to the memory controller where the 3D structure of modern DRAM is exploited to increase throughput.

Traditionally, prefetching reduces latency by fetching data before it is needed. In this paper we explore how prefetching can be used to increase memory throughput. We present our own low-cost open-page prefetch scheduler that exploits the 3D structure of DRAM when issuing prefetches. We show that because of the complex structure of modern DRAM, prefetches can be made cheaper than ordinary reads, thus making prefetching beneficial even when prefetcher accuracy is low. As a result, prefetching with good coverage is more important than high accuracy. By exploiting this observation our low-cost open page scheme increases performance and QoS. Furthermore, we explore how prefetches should be scheduled in a state of the art memory controller by examining sequential, scheduled region, CZone/Delta Correlation and reference prediction table prefetchers.

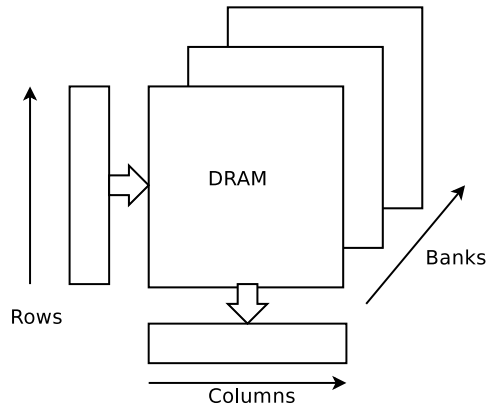


Figure F.1: The 3D structure of modern DRAM.

F.1 Introduction

Chip Multiprocessors have been introduced by virtually all makers of high performance processors. CMPs shifts the focus away from the traditional uniprocessor paradigm, where low latency and instruction-level parallelism (ILP) is important to a paradigm where throughput and thread-level parallelism (TLP) dominates.

This shift is reflected in the memory subsystem as well, where the memory controllers have traditionally been used to reduce system latency. However, as more cores are added to a chip, off-chip bandwidth are shared across cores, thus increasing the pressure on this resource and lowering locality in the memory access stream. Thus, memory controllers have been designed to optimize for maximum throughput, at the expense of increasing worst-case latency.

This increase in throughput has been made possible by exploiting the 3D structure of modern DRAM [4]. DRAM is organized in several banks. Each bank is organized as a matrix of rows and columns of DRAM cells as shown in figure F.1. In a normal read operation, a bank and a row is first selected for activation. The charges from this row of capacitors are then amplified by sense-amplifiers in the DRAM module and stored in a large latch. Each such row is commonly referred to as a page. A page is normally about 1KB to 4KB large, whereas a cacheline is typically 64-256B large. Thus, a page will typically hold several consecutive cachelines. The portion of the page that was requested is then transferred over the data-bus. When the page is no longer needed, the memory controller instructs the DRAM module to write the latch contents back into the DRAM cells, preserving the contents of the page. This is referred to as closing the page.

In terms of latency, opening and closing a page is expensive, while getting data out of the latches and over the data-bus is comparatively cheap. In addition, there is a minimum allowed time between opening and closing a page (the minimum activate-

to-precharge latency). Thus a single read is slow, but reading the next cache block is relatively cheap as the page is already open and the data is in the latch. This property is exploited by the First Ready, First Come, First Served (FR-FCFS) memory controller proposed by Rixner et al. [16]. This type of memory controller allows accesses that uses an already open page to be scheduled even if the request is not the oldest.

Traditionally, prefetching has been used to decrease latency for a single operation by speculatively bringing data into the cache before it is needed. In this paper we exploit the 3D structure of modern DRAM to demonstrate how prefetching can be used to increase off-chip bandwidth utilization. Because there is a lower cost associated with fetching data that resides in an open page, we prefetch this data, provided that our confidence that the data will be useful is high enough. In addition, we show that prefetching can be effective at relatively low accuracy, due to the low cost of piggybacking prefetches compared to single reads. Finally, we present our low cost open page prefetching scheduling heuristic which exploits this observation.

F.2 Previous Work

F.2.1 Prefetching

Previously, Wei-Fen et al. [9] have examined how prefetches can be scheduled in a uniprocessor context with Rambus DRAM. They used a dedicated prefetch queue with a LIFO insertion policy with a scheduled region prefetcher. In addition, Cantin et al. [2] exploited open pages to increase the performance of their stealth prefetcher.

There exists a multitude of different prefetching schemes. The simplest is the *sequential prefetcher* [18], which simply fetches the next block whenever a block is referenced. However, more complex types exists as well, such as the *CZone/Delta Correlation (C/DC)* prefetcher proposed by Nesbit et al. [12, 13]. C/DC divides memory into CZones and analyses patterns contained in the reference stream by using a Global History Buffer (GHB) to store recent misses to the cache. Lin et al. [8] introduced *scheduled region prefetching* (SRP) which issues prefetches to blocks spatially near the addresses of recent demand missed when the memory channel is idle. Other types, such as the *Reference Prediction Table Prefetcher* (RPT) proposed by Chen and Baer [3] examines the pattern generated by a load instruction with a state machine. Somogyi et al. proposed *Spatial Memory Streaming* (SMS) [19]. SMS uses code-correlation to predict spatial access patterns.

F.2.2 Memory Controllers

Memory access scheduling is the process of reordering memory requests to improve memory bus utilization. Rixner et al. [16] showed that significant speed-ups are possible when memory request reordering is applied to stream processors. In addition, Shao et al. [17] proposed *burst scheduling* in which multiple read and write requests to the same DRAM page are issued together to achieve high bus utilization. Finally, Zhu et al. [24] showed that it is beneficial to divide the memory requests into smaller parts, and give priority to the words responsible for a processor stall in a multi-channel DRAM system.

CMPs, processors with SMT support and conventional shared-memory multiprocessors also benefit from memory access scheduling. Zhu et al. [23] showed that DRAM throughput could be increased in an SMT processor by using ROB and IQ occupancy status to prioritize requests. Furthermore, Hur et al. [5] use a history-based arbiter to adapt the DRAM port and rank schedule to the application's mix of reads and writes for the dual-core Power5 processor. In addition, Natarajan et al. [11] showed that a significant performance improvement is available by exploiting memory controller features in a conventional, shared-memory multiprocessor.

In CMPs, the memory bus is shared between all processing cores and a number of researchers have looked into how this can be accomplished in a fair way [6, 10, 14, 15]. In general, bandwidth is divided among threads according to their priorities at the same time as requests are scheduled in a way that improves DRAM throughput.

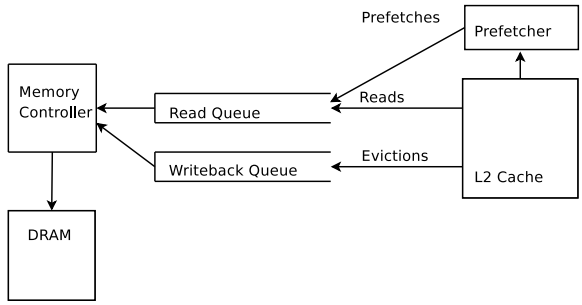
F.3 Prefetch Scheduling

A prefetching heuristic can be characterized by using two distinct metrics: *Accuracy* is a measure of how many of the issued prefetches have actually been useful to the processor [22], while coverage measures how many of the potential prefetches have been issued.

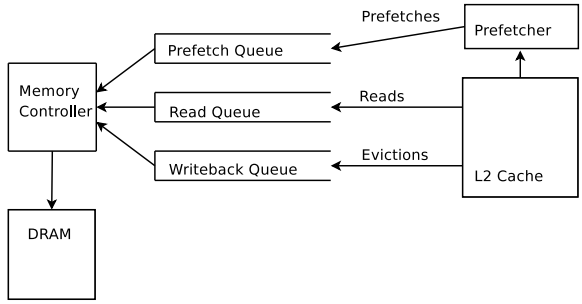
Because prefetching is a speculative technique, there are two potential sources for performance degradation. Firstly, prefetching consumes additional bandwidth as some data transferred over the memory bus is not used. Secondly, it can pollute the cache, by displacing data that is still needed.

The FR-FCFS memory scheduler [16] is a high throughput memory scheduler. It exploits the 3D structure of modern DRAM by allowing requests that would access an already open page to bypass the normal FCFS queue. FR-FCFS prioritizes memory requests in the following manner: 1) Ready operations (operations that access open pages), 2) CAS (column selection) over RAS (row selection) commands, and 3) Oldest request first. In addition, reads have a higher priority than writes.

There are two basic ways to introduce prefetching into the FR-FCFS memory controller. The simplest approach is to insert prefetch requests into the read queue, as



(a) Conventional prefetch scheduling



(b) Dedicated prefetch queue

Figure F.2: Prefetch scheduling policies

shown in figure F.2(a). A more sophisticated approach introduced by Lin Wei-Fen et al. [9] is to use a dedicated queue for prefetches as shown in figure F.2(b). In this approach, prefetches are prioritized after writebacks, so the priority rule becomes: Prioritize read operations over writeback operations over prefetch operations.

F.4 Low cost open page prefetching

After a demand read to DRAM is serviced, the page that the demand read resided in is still open, and in most cases cannot be closed due to the minimum activate to precharge latency. Other DRAM banks can still be utilized. If a prefetch or read is issued to this open page, there is little latency as the data requested is already in the latch. In this paper we refer to this as *piggybacking*.

By allowing prefetches to piggyback on regular read requests, the cost of prefetching is effectively reduced. In the dedicated prefetch queue approach, prefetches are only issued if they can piggyback on another request, or if the bus is idle. Suppose a processor requires data at locations X_1 and X_2 that are located on the same page at times T_1 and T_2. There are two separate outcomes: If T_1 and T_2 are sufficiently close, both requests will be in the memory controller at the same time, and request

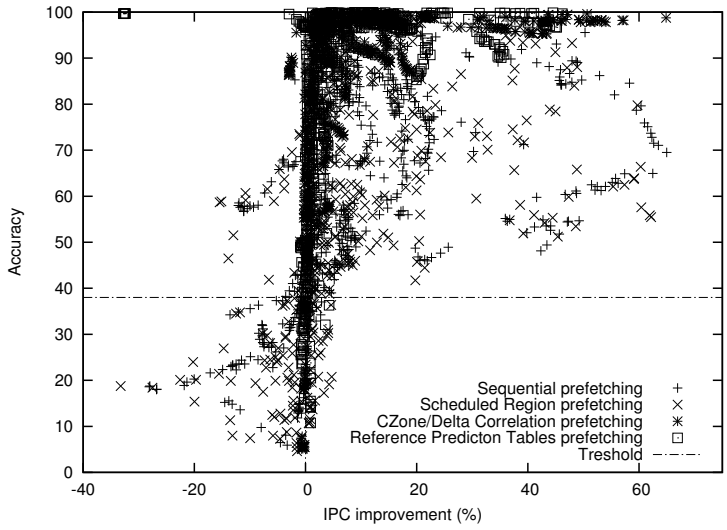


Figure F.3: IPC improvement as a function of accuracy

2 can piggyback on request 1. Thus the page only needs to be opened once. If the two requests are sufficiently separated in time, the two requests cannot be piggybacked on each other, thus forcing the page to be opened twice. This reduces overall throughput. In the second case, prefetching X_2 can increase performance by both reducing latency and increase memory throughput. However, because prefetching is a speculative technique, its prediction for what data is needed in the future might be wrong. Thus, there is a break-even point where the benefit of prefetching is balanced against the cost of prefetching.

To test this assumption we have conducted experiments on 4 different prefetching heuristics (Sequential, SRP, C/DC and RPT) with 10 different prefetching configurations (each) on 40 different workloads. We measured the accuracy of the prefetcher and the IPC improvement (versus a configuration with no prefetching). Our results are shown in figure F.3.

In this graph it is clear that most of the points fall into 2 quadrants. One where accuracy is below 38% and performance is decreased, while another where accuracy is above 38% and performance is increased.

Our prefetch scheduler exploits this observation by measuring prefetch accuracy at runtime. If the accuracy falls below a treshold (in our experiments 38%) then prefetches are no longer piggybacked on open pages and only issued if the bus is idle.

We use an accuracy estimator similar to the one used by Sriniath et al. [21]. When a prefetch is issued, a counter is increased (indicating the number of prefetches issued) and a prefetched-bit is set in the corresponding cache line. This bit is already

Table F.1: Processor Core Parameters

Parameter	Value
Processor Cores	4
Clock frequency	3.2 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 4-way 2048 entry BTB

Table F.2: Memory System Parameters

Parameter	Value
Level 1 Data Cache	64 KB, 8-way set associative, 64B blocks, 3 cycles latency
Level 1 Instruction Cache	64 KB, 8-way set associative, 64B blocks, 1 cycle latency
Level 2 Unified Shared Cache	4 MB, 16-way set associative, 64B blocks, 14 cycles latency, 8 MSHRs per bank, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 9 cycles latency, 64B wide transmission channel
DDR2 memory	400 Mhz Clock, 8 banks, 1KB pagesize, 4-4-4-12 timing, dual channel in lock-step

present when using sequential prefetching, and thus causes no additional overhead. The first time a cache line with this bit set is referenced by the program, the bit is cleared and another counter (indicating the number of successful prefetches) is increased. By sampling the successful prefetch counter every time the 10 bit issued counter wraps, we get an estimate of the prefetchers accuracy.

F.5 Methodology

We used the system call emulation mode of the cycle-accurate M5 simulator [1] to evaluate our scheme. The processor architecture parameters for the simulated 4-core CMP are shown in table F.1, and table F.2 contains the baseline memory system parameters. We have extended M5 with a crossbar interconnect, a detailed DDR2 memory bus and DRAM model, a FR-FCFS memory controller and prefetching.

Our DDR2-implementation [7] models separate RAS, CAS and precharge commands. In addition, we model pipelining of requests, independent banks, burst mode transfers and bus contention. The FR-FCFS memory controller has a 128 entry read-queue, 64 entry writeback queue and a 128 entry prefetch queue. As the conventional method of issuing prefetches has no separate prefetch queue, the read queue has been increased to 256 entries to make comparison more fair in terms of area. Unless otherwise noted, we use 4KB regions in scheduled region prefetching, 256KB CZones, a 1024-entry global history buffer and a 16-entry reference prediction table.

The SPEC CPU2000 benchmark suite [20] is used to create 40 multiprogrammed workloads consisting of 4 SPEC benchmarks each as shown in table F.3. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the random selection process was that each SPEC benchmark had to be represented in at least one workload. To avoid unrealistic interference when more than a single instance of a benchmark is part of a workload, the benchmarks are fast-forwarded a random number of clock cycles between 1 and 1.1 billion. Then, detailed simulation is carried out for 100 million clock cycles measured from the clock cycle the last core finished fast forwarding. As our metric of throughput we have used the average IPC of all 4 cores. In most cases, performance is measured as the relative increase in speed compared to the no prefetching case.

F.6 Results

F.6.1 Scheduled Region Prefetching

In figure F.4 we show the relative performance of each of the prefetch scheduling policies. In this experiment we use a scheduled region prefetcher (SRP) with 4KB regions. The conventional and dedicated prefetch queue options give an average of 14.4% increase in performance versus the no prefetching case, while the average increase for our scheme is 17.1%. In addition, prefetching causes performance degradation in 9 out of the 40 cases. Our prefech scheduling policy reduces the performance penalty on 6 of these workloads. However, a lot of information is lost in averages. For instance, the performance increased on workload 1 is only 1% in other schemes, while our method increases performance by 15%. Similar results can be seen in workload 6, 7, 23, 25, 27, 28, 32 and 38.

F.6.2 Importance of Coverage

In figure F.5 we show the average relative performance increase by using other types of prefetchers, including Scheduled Region Prefetching, CZone/Delta Correlation and Reference Prediction Tables. Both C/DC and RPT prefetching have high accuracy. Because yhe prefetching accuracy is higher than the treshold in almost all workloads, our method degrades into the dedicated prefetch queue. In turn, the performance of our prefetch scheduling scheme is almost equal to the dedicated prefetch queue scheme. However, this graph shows another interesting property. Scheduled Region Prefetching, which has a comparatively low accuracy, outperforms both of the more complex prefetcher heuristics. This is due to it having a much higher prefetch coverage. It provides more prefetches with *acceptable* accuracy, thus increasing performance.

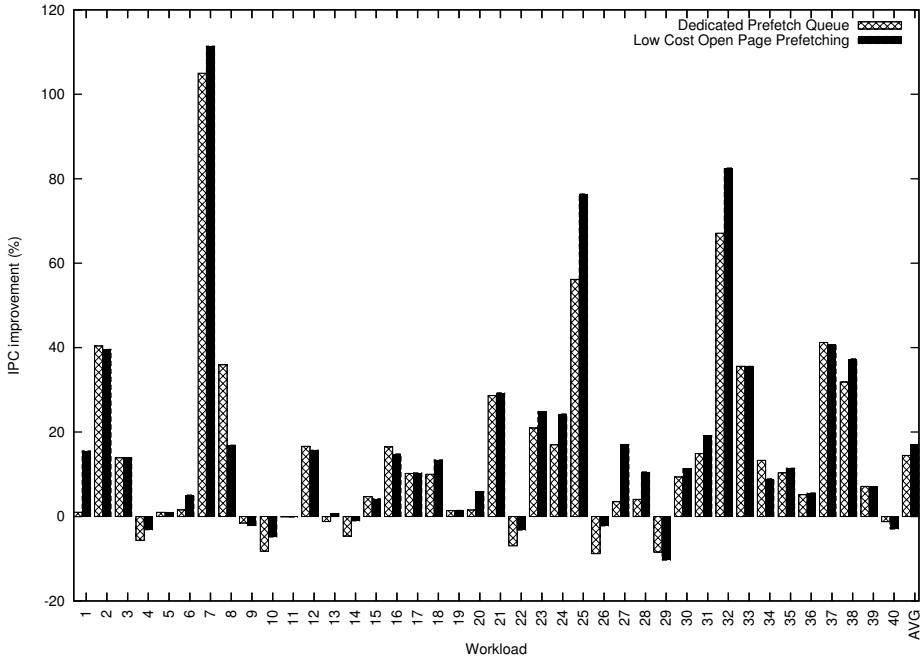


Figure F.4: Speedup in IPC relative to no prefetching using a FR-FCFS memory controller.

F.6.3 Insertion policy

In our scheme and the dedicated prefetch queue scheme there is a separate queue for handling prefetches. There are multiple possibilities on how to insert new prefetches into the queue. If the prefetch queue is full, then there are two possibilities, either discard the prefetch or insert the prefetch and evict the oldest prefetch. In figure F.6 we show the performance of FIFO and LIFO policies with and without evictions. From this graph it is clear that evicting old data is beneficial, as well as using a LIFO policy. Evicting old prefetches is useful, because newer prefetches are based on newer demand reads, thus increasing both the accuracy and the probability that it can be piggybacked. The LIFO policy ensures that the newest prefetches are given priority over old ones. As shown in the graph, for both techniques, evicing old data is preferable, while a LIFO policy gives marginally better results over FIFO.

F.6.4 Treshold parameter

In figure F.7 we show the average speedup as a function of the required accuracy (treshold). In effect, setting the treshold to 0% makes the low cost open page

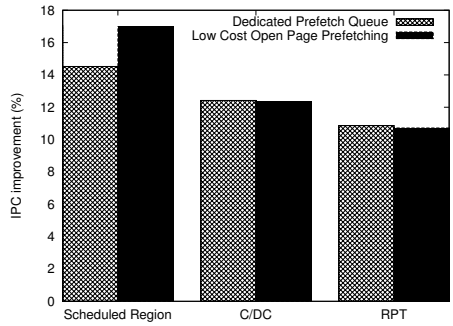


Figure F.5: Average speedup in IPC relative to no prefetching.

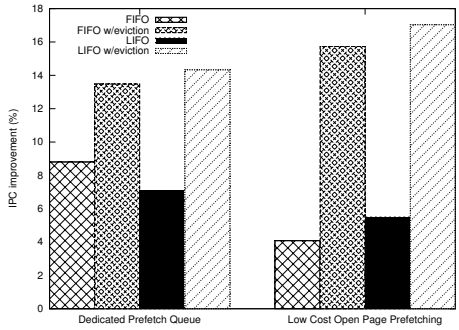


Figure F.6: Effects of insertion policy on average IPC speedup.

prefetcher a dedicated queue prefetcher. Both RPT and C/DC prefetching have a very high accuracy, so the treshold doesn't affect performance until it becomes too high, effectively disabling prefetching, and in turn degrades performance. In addition, the peak for both sequential and scheduled region prefetching is relatively low (around 20-30 %). This further supports the observation that coverage is more important as long as accuracy is acceptable.

F.6.5 Quality of Service

We have measured the maximum slowdown for any thread compared to the case where no prefetching is performed on each workload to get an indicator of the quality of service. Figure F.8 shows the maximum performance degradation as a function of the number of workloads included. This graph shows three important properties. Firstly, 25% of the workloads experience no performance degradation on any thread when doing prefetching. Secondly, our scheme gives consistently higher quality of service. Using the other scheme 33% of the workloads show a thread getting a performance degradation of above 10%. In our scheme only 20%

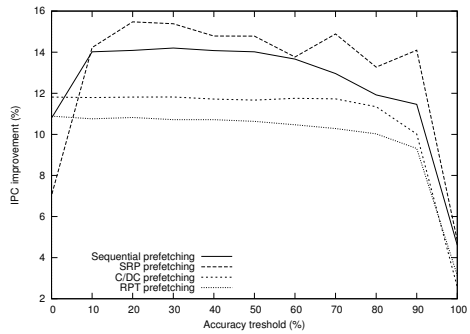


Figure F.7: IPC improvement as a function of treshold

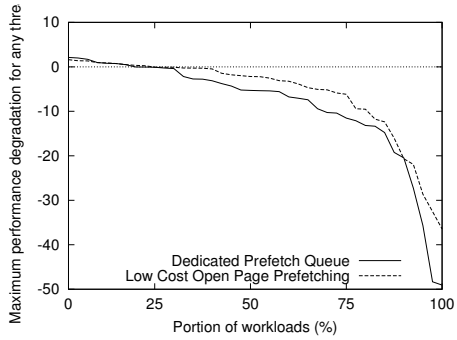


Figure F.8: Maximum IPC degradation for any thread as a function of workloads.

of the workloads show a thread getting more than 10% performance degradation. Finally, the maximum degradation for any thread for our scheme is only 36%, while the maximum for the dedicated prefetch queue approach is 49%.

F.7 Discussion

Our results show that it is more important to have good prefetching coverage, while having *acceptable* accuracy. This is due to the relatively lower cost of piggybacked prefetches compared to isolated demand reads. Normally prefetch heuristics have been optimized for maximizing accuracy, so that the impact on bandwidth is as low as possible. This is due to the assumption that the cost of a single prefetch is about the same as a demand read. By carefully scheduling prefetches so that they are piggybacked on normal demand reads, this assumption no longer holds. We have demonstrated that a simpler, high coverage prefetcher outperforms more sophisticated high accuracy prefetchers in a bandwidth-constrained, 4-core chip multiprocessor system.

Table F.3: Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	ammp, mgrid, perlbmk, parser	11	vpr, twolf, applu, eon	21	perlbmk, apsi, lucas, equake	31	mgrid, equake, vpr, eon
2	lucas, gcc, mcf, twolf	12	galgel, crafty, mgrid, swim	22	vpr, crafty, vpr, mcf	32	wupwise, gap, twolf, facerec
3	eon, eon, mesa, facerec	13	twolf, fma3d, galgel, vpr	23	gzip, equake, mgrid, mesa	33	galgel, equake, lucas, gzip
4	vortex1, ammp, equake, galgel	14	bzip, vpr, bzip, equake	24	facerec, applu, fma3d, lucas	34	facerec, gcc, facerec, apsi
5	gcc, galgel, apsi, crafty	15	galgel, crafty, vpr, swim	25	gap, applu, parser, facerec	35	mesa, mcf, swim, sixtrack
6	applu, equake, art, facerec	16	mcf, wupwise, mesa, mesa	26	mcf, apsi, twolf, ammp	36	mesa, sixtrack, equake, bzip
7	applu, gap, gcc, parser	17	applu, parser, apsi, perlbmk	27	swim, sixtrack, ammp, applu	37	mcf, gap, gcc, vortex1
8	gap, swim, twolf, mesa	18	mgrid, perlbmk, gzip, mgrid	28	art, fma3d, swim, parser	38	facerec, lucas, mcf, parser
9	sixtrack, fma3d, apsi, vortex1	19	mcf, sixtrack, gcc, apsi	29	apsi, gcc, vortex1, twolf	39	twolf, eon, mesa, eon
10	ammp, bzip, equake, parser	20	ammp, gcc, art, mesa	30	mgrid, gzip, apsi, equake	40	apsi, apsi, mcf, equake

In our prefetch scheduling heuristic, we have used an accuracy estimator to control when prefetches should be issued. Other researchers have used such an estimator to control the aggressiveness of the prefetcher [21]. Such a technique can be used in conjunction with our scheduler. By using a feedback directed prefetcher, coverage can be increased while keeping accuracy at an acceptable level, thus providing higher performance.

Our simulator does not include a power model. However, our scheme piggybacks prefetches on demand reads. If a prefetch is successful then a later read is not needed, thus reducing the number of pages opened and closed, which in turn reduces power consumption in the DRAM module. Prefetching invariably increases bus traffic as some data transferred is not needed. Our scheme reduces the amount of useless traffic compared to other schemes by filtering out prefetches with low accuracy, thereby saving power.

F.8 Conclusion

In this paper we have shown that by carefully scheduling prefetches so that they piggyback on ordinary demand reads, performance can be increased. This is done by exploiting the 3D structure of modern DRAM, where opening and closing pages is an expensive operation. As it becomes more important to issue prefetches that

can be piggybacked on ordinary demand reads, emphasis shifts from high accuracy to high coverage with *acceptable* accuracy.

We have demonstrated our own prefetch scheme on a state of the art memory controller that exploits these findings. Our prefetch policy outperforms traditional scheduling policies in terms of performance, quality of service and power consumption.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth Prefetching. *SIGPLAN Notices*, 41(11), 2006.

[3] T. Chen and J. Baer. Effective Hardware-Based Data Prefetching for High-performance Processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

[4] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proc. of the 26th Inter. Symp. on Comp. Arch.*, pages 222–233, 1999.

[5] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *MICRO 37: Proc. of the 37th An. IEEE/ACM Int. Symp. on Microarch.*, pages 343–354, 2004.

[6] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS '07*, pages 25–36, 2007.

[7] *DDR2 SDRAM Specification.* JEDEC Solid State Tech. Association, May 2006.

[8] W. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[9] W. Lin, S. K. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching. *IEEE Transactions on Computers*, 50(11), 2001.

[10] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Int. Symp. on Microarchitecture*, 2007.

[11] C. Natarajan, B. Christenson, and F. Briggs. A Study of Performance Impact of Memory Controller Features in Multi-processor Server Environment. In *WMPI '04: Proc. of the 3rd Workshop on Memory Perf. Issues*, pages 80–87, 2004.

[12] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. *IEEE Micro*, 25:90–97, 2005.

[13] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An Adaptive Data Cache Prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[14] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *MICRO 39: Int. Symp. on Microarchitecture*, pages 208–222, 2006.

[15] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[16] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[17] J. Shao and B. Davis. A Burst Scheduling Access Reordering Mechanism. In *HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch.*, 2007.

[18] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[19] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. *SIGARCH Computer Architecture News*, 34(2):252–263, 2006.

[20] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[21] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. Technical report, University of Texas at Austin, May 2006. TR-HPS-2006-006.

[22] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A Prefetch Taxonomy. *IEEE Transactions on Computers*, 53:126–140, 2004.

[23] Z. Zhu and Z. Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *HPCA '05: Proc. of the 11th Int. Symp. on High-Performance Comp. Arch.*, pages 213–224, 2005.

[24] Z. Zhu, Z. Zhang, and X. Zhang. Fine-Grain Priority Scheduling on Multi-channel Memory Systems. *8th Int. Symp. on High-Performance Comp. Arch.*, pages 107–116, 2002.

Appendix G

Paper C.II

Exploring the Prefetcher/Memory Controller Design Space: An Opportunistic Prefetch Scheduling Strategy

Marius Grannæs, Magnus Jahre and Lasse Natvig
Submitted to *Journal of Computer Science and Technology*
2010

Abstract

Prefetching is a well-known technique for bridging the memory gap. By predicting future memory references the prefetcher can fetch data from main memory and insert it into the cache such that overall performance is increased. Modern memory controllers reorder memory requests to exploit the 3D structure of modern DRAM interfaces. In particular, prioritizing memory requests that use open pages increases throughput significantly.

In this work, we investigate the prefetcher/memory controller design space along three dimensions: prefetching heuristic, prefetch scheduling strategy and available memory bandwidth. In particular, we evaluate 5 different prefetchers and 6 prefetch scheduling strategies. Through this extensive investigation, we observed that prior prefetch scheduling strategies often cause memory bus contention in bandwidth constrained CMPs which in turn causes performance regressions. To avoid this problem, we propose a novel prefetch scheduling heuristic called *Opportunistic Prefetch Scheduling* that selectively prioritizes prefetches to open DRAM pages such that performance regressions are minimized. Opportunistic prefetch scheduling reduces performance regressions by 6.7X and 5.2X, while improving performance by 17 % and 20 % for sequential and scheduled region prefetching, compared to the direct scheduling strategy.

G.1 Introduction

The pressure on off-chip memory increases significantly as more cores compete for the same resources. A CMP deals with the memory wall by exploiting thread level parallelism (TLP), shifting the focus from reducing overall memory latency to memory throughput. This extends to the memory controller where the 3D structure of modern DRAM (Figure G.1) is exploited to increase throughput. Because of this 3D structure, the latency of a memory operation varies depending on bank conflicts and open pages. In particular, fetching data from open pages is beneficial as it has low latency and increases DRAM throughput [14].

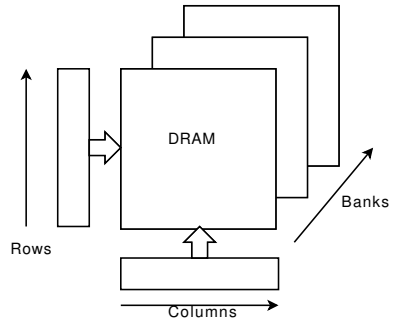


Figure G.1: 3D structure of DRAM.

Prefetching reduces latency by fetching data before it is needed. Research has shown that prefetching can also increase bandwidth utilization by tight integration of the prefetcher with the memory controller [4, 8]. This is achieved through prioritizing prefetches based on prefetcher accuracy, available memory bandwidth and interaction with open pages. Fetching data from an open page has a much lower cost in latency than a complete cycle of opening a page, fetching the data and closing the page. Because of this difference in cost, speculatively prefetching data from an open page is beneficial even at relatively low prefetcher accuracy [4]. However, this is a trade-off between latency and the available bandwidth.

Because of the complex interaction between the prefetcher and the memory controller, prefetch scheduling can be as important as determining which addresses to prefetch. In this work, we investigate the prefetcher/memory controller design space along three dimensions: data prefetching heuristic, prefetch scheduling strategy and available memory bandwidth. In particular, we evaluate 5 different prefetchers and 6 prefetch scheduling strategies. Through this extensive investigation, we observed that prior prefetch scheduling strategies often cause memory bus contention in bandwidth constrained CMPs, which in turn causes performance regressions. To avoid this problem, we propose a novel prefetch scheduling heuristic called *Opportunistic Prefetch Scheduling* that only issues prefetches to open DRAM pages such that performance regressions is minimized.

This strategy has to major effects: The drawback is that prefetching opportunities might be lost, because a demand read is always required prior to prefetching to a new DRAM page. This reduces overall performance when prefetch accuracy is high and there is ample bandwidth. The advantage of this strategy is that because prefetches cannot open new pages, a prefetch cannot delay a demand read by opening and closing pages. Overall, this makes opportunistic prefetching a defensive strategy, which sacrifices some performance in an effort to minimize performance regressions in cases where the prefetcher performs badly.

This scheduling strategy also bridges the performance gap between the simpler prefetchers (e.g. sequential) and the more complex prefetchers in future multicore architectures where there will be more contention for off-chip bandwidth. Using simple prefetchers is an advantage because they are easier to implement and verify.

G.2 Related Work

G.2.1 Prefetching

There exists a multitude of different data prefetching schemes. The simplest is the *sequential prefetcher* [16], which simply fetches the next block whenever a block is referenced. However, more complex types exists as well, such as the *CZone/Delta Correlation (C/DC)* prefetcher proposed by Nesbit et al. [12, 13]. C/DC divides memory into CZones and analyses patterns contained in the reference stream by using a Global History Buffer (GHB) to store recent misses to the cache. Lin et al. [9] introduced *scheduled region prefetching* (SRP) which issues prefetches to blocks spatially near the addresses of recent demand missed when the memory channel is idle. Other types, such as the *Reference Prediction Table Prefetcher* (RPT) proposed by Chen and Baer [3], examines the pattern generated by a load instruction with a state machine. *Delta Correlating Prediction Table* (DCPT) is a table based approach which stores the history of each load instruction in the form of address deltas [5]. DCPT prefetches new data by using delta correlation to find patterns in the history of deltas. Somogyi et al. proposed *Spatial Memory Streaming* (SMS) [17]. SMS uses code-correlation to predict spatial access patterns.

G.2.2 Memory Controllers

Memory access scheduling is the process of reordering memory requests to improve memory bus utilization. Rixner et al. [14] showed that significant speed-ups are possible when memory request reordering is applied to stream processors. In addition, Shao et al. [15] proposed *burst scheduling* in which multiple read and write requests to the same DRAM page are issued together to achieve high bus utilization. Finally, Zhu et al. [20] showed that it is beneficial to divide the memory requests

into smaller parts, and give priority to the words responsible for a processor stall in a multi-channel DRAM system.

CMPs, processors that support SMT (Simultanous Multi-Threading) and conventional shared-memory multiprocessors also benefit from memory access scheduling. Zhu et al. [19] showed that DRAM throughput could be increased in an SMT processor by using ROB (ReOrder Buffer) and IQ (Instruction Queue) occupancy status to prioritize requests. Furthermore, Hur et al. [6] use a history-based arbiter to adapt the DRAM port and rank schedule to the application's mix of reads and writes for the dual-core Power5 processor. In addition, Natarajan et al. [11] showed that a significant performance improvement is available by exploiting memory controller features in a conventional, shared-memory multiprocessor.

G.3 Prefetch Scheduling Strategies

Earlier work have focused on the interaction between a specific prefetcher and the memory controller. Wei-Fen et al. [10] have examined how prefetches can be scheduled in a uniprocessor context with Rambus DRAM. They used a dedicated prefetch queue with a LIFO insertion policy with a scheduled region prefetcher. Cantin et al. [2] exploited open pages to increase the performance of their stealth prefetcher. In this work, we decouple the prefetching scheduling strategy from the prefetcher and examine new combinations of prefetcher and prefetch scheduling strategy. This allows us to do a design space exploration where we examine many combinations of prefetchers and prefetch scheduling strategies.

The simplest way to schedule prefetches in a memory controller is to treat them as demand reads. This method requires no additional infrastructure and most controllers can easily accommodate this technique. Because the prefetches are treated as reads, they cannot be discarded by the memory controller which in turn can lead to memory congestion and memory controller blocking due to full queues. In this paper, we refer to this strategy as the *Direct* prefetch scheduling strategy.

This situation can be improved by adding an additional queue called the *prefetch queue* [2, 8, 10]. A dedicated prefetch queue can hold prefetches separate from the reads which enables the memory controller to selectively issue or discard prefetches. Because prefetches can be discarded, the memory controller can choose to discard prefetches in the prefetch queue rather than block. However, because there are now two (in addition to the write queue) queues, a method for choosing which queue to issue reads or prefetches from is needed.

The most restrictive method is to only issue prefetches from the dedicated prefetch queue when there are no other operations pending. We refer to this as the *Idle* prefetch scheduling strategy. A more aggressive approach is to schedule prefetches when any of the prefetches currently in the queue would read data from a currently open page. This is often beneficial because a prefetch into an open page costs less than a demand read. In this paper, we refer to this strategy as the *Ready* prefetch

Page Address	Bit Vector															
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
100	00	01	00	01	00	01	00	01	00	01	00	01	00	01	00	01
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Table G.1: Example Page Vector Table showing a strided prefetch pattern for page address 100.

scheduling strategy. Because the dedicated prefetch queue has a finite size, a policy for clearing space for new prefetches is needed when the queue is full. In this paper, we simply drop the oldest prefetch request. This method has been shown to be the most effective policy as it removes the least timely prefetches [4].

The accuracy of the prefetcher can be measured at runtime by tagging each prefetched cache block with a prefetch bit. This bit is set when a cache block is inserted by a prefetch. The first time the cache block is accessed, the bit is cleared and the "good prefetches" counter is incremented. Similarly, when a prefetch is issued by the memory controller the "prefetches issued" counter is incremented. The ratio between these two counters is the estimated accuracy.

Grannaes et al. used this estimated accuracy to switch between the *Idle* and the *Ready* prefetch scheduling strategy [4]. If the estimated accuracy was below 40%, the *Idle* strategy was used. If it was over 40% the *Ready* strategy was used. In this work, we refer to this strategy as the *Low Cost* strategy.

Lee et al. used the estimated accuracy to switch between the *Ready* and the *Direct* scheduling strategies [8]. When the accuracy was estimated to be high, the *Direct* strategy was used. Conversely, the *Ready* strategy was used when the accuracy was estimated to be low. In this paper, we refer to this strategy as the *Hybrid* strategy.

G.3.1 Opportunistic Prefetch Scheduling

In this paper, we propose a prefetch scheduling strategy called *Opportunistic Prefetch Scheduling* which is strongly tied to the observation that prefetching from open pages is beneficial. Instead of using a queue of prefetches, we use a *Page Vector Table* (PVT) indexed by the page-address. An example of a PVT is shown in Table G.1. In this paper, we use a 256 entry PVT with a LRU replacement policy.

The page-address is the portion of the memory address which maps to physical DRAM pages. In our setup, each DRAM page is 1KB large and each cache block is 64B large. Each PVT entry consists of a 16 entry two-bit vector where each pair of bits represents one cache block in the page. The two bits represents demand reads and prefetches (demand bit and prefetch bit). When the memory controller issues a demand read for a cache block, it marks the corresponding demand read bit in the vector for that page. When the prefetcher generates a prefetch for a cache block, it looks up the DRAM page in the table and sets the corresponding prefetch bit in the vector for that page.

We assume a FR-FCFS memory controller with a closed page policy (idle page close). The PVT is consulted before the memory controller closes any page (e.g. after all demand reads have been serviced). If the prefetch bit is set, but not the demand read bit, then a prefetch is issued for the corresponding address. This approach ensures that prefetches are only issued if they access an open page. The bitvector representation is very compact as only two bits are stored per prefetch (excluding the page tag) compared to the traditional approach using queues which holds the full address.

To reduce the potential for performance regressions, *Opportunistic* estimates the accuracy of the issued prefetches to choose one of two substrategies. If accuracy is high, prefetches are issued when a page is closed. If accuracy is low, prefetches are only issued when a page is closed and there are no demand reads in the queue.

The end result of using the opportunistic prefetch scheduling strategy is that pages are never opened due to prefetching. This strategy has to major effects: The drawback is that prefetching opportunities might be lost, because a demand read is always required prior to prefetching to a new DRAM page. This reduces overall performance when prefetch accuracy is high and there is ample bandwidth. The advantage of this strategy is that because prefetches cannot open new pages, a prefetch cannot delay a demand read as much due to the opening and closing of pages. Overall, this makes opportunistic prefetching a defensive strategy, which sacrifices some performance in an effort to minimize performance regressions in cases where the prefetcher performs badly.

G.4 Methodology

To examine the impact of different prefetch scheduling strategies we have used the M5 simulator [1]. The processor architecture parameters for the simulated 4-core CMP are shown in Table G.2, and Table G.3 contains the baseline memory system parameters. We have extended M5 with a crossbar interconnect, a detailed DDR2 memory bus and DRAM model and a detailed FR-FCFS (First Ready, First Come, First Served) [14] memory controller with integrated prefetching capabilities.

Our DDR2-implementation [7] models separate RAS, CAS and precharge commands. In addition, we model pipelining of requests, independent banks, burst mode transfers and bus contention. All prefetchers use a prefetching degree of 10. We use 1KB regions in scheduled region prefetching, 256KB CZones, a 1024-entry global history buffer and a 16-entry reference prediction table. DCPT uses a 128 entry table with each entry holding 20 18-bit entries. The Page Vector Table consists of 256 32 bit vectors.

The SPEC CPU2000 benchmark suite [18] is used to create 40 multiprogrammed workloads consisting of 4 SPEC benchmarks each as shown in Table G.4. We picked benchmarks at random from the full SPEC CPU2000 benchmark suite, and each processor core is dedicated to one benchmark. The only requirement given to the

Table G.2: Processor Core Parameters

Parameter	Value
Processor Cores	4
Clock frequency	3.2 GHz
Reorder Buffer	128 entries
Store Buffer	32 entries
Instruction Queue	64 instructions
Instruction Fetch Queue	32 entries
Load/Store Queue	32 instructions
Issue Width	8 instructions/cycle
Functional units	4 Integer ALUs, 2 Integer Multipy/Divide, 4 FP ALUs, 2 FP Multiply/Divide
Branch predictor	Hybrid, 2048 local history registers, 4-way 2048 entry BTB

Table G.3: Memory System Parameters

Parameter	Value
Level 1 Data Cache	64 KB, 8-way set associative, 64B blocks, 3 cycles latency
Level 1 Instruction Cache	64 KB, 8-way set associative, 64B blocks, 1 cycle latency
Level 2 Unified Shared Cache	4 MB, 16-way set associative, 64B blocks, 14 cycles latency, 8 MSHRs per bank, 4 banks
L1 to L2 Interconnection Network	Crossbar topology, 9 cycles latency, 64B wide transmission channel
DDR2 memory	400 Mhz Clock, 8 banks, 1KB pagesize, 4-4-4-12 timing, dual channel in lock-step
Memory Controller	128 entry queue, Ready First - First Come, First Served policy for reads

random selection process was that each SPEC benchmark had to be represented in at least one workload. Each workload is first fast forwarded 1 billion clock cycles and then detailed simulation is carried out for 100 million clock cycles.

G.5 Results

G.5.1 Performance

Figure G.2 shows the average speedup of each prefetch scheduling strategy with five different prefetchers (Sequential, RPT, C/DC, SRP and DCPT) in a system with one DRAM channel. *Opportunistic* performs well in combination with both the sequential and the SRP prefetcher. For RPT, C/DC and DCPT, the *Ready* and *Low cost* strategies performs slightly better. However, *Opportunistic* prefetch scheduling is able to bridge the performance gap between the simple sequential prefetcher and the more complex RPT, CDC and DCPT prefetchers. This is because the overall accuracy of these prefetchers is typically higher than for sequential and SRP prefetching. This same effect can be observed for the *Direct* strategy where performance is low for the sequential and SRP prefetcher while it is higher for the other prefetchers. This is because the strategy does not make any distinction between prefetches and demand reads. Thus, inaccurate prefetches can disrupt demand reads. The performance for the *Idle* strategy is comparably low for most

Table G.4: Multiprogrammed Workloads

ID	Benchmarks	ID	Benchmarks	ID	Benchmarks	ID	Benchmarks
1	ammp, mgrid, perlbmk, parser	11	vpr, twolf, applu, eon	21	perlbmk, apsi, lucas, equake	31	mgrid, equake, vpr, eon
2	lucas, gcc, mcf, twolf	12	galgel, crafty, mgrid, swim	22	vpr, crafty, vpr, mcf	32	wupwise, gap, twolf, facerec
3	eon, eon, mesa, facerec	13	twolf, fma3d, galgel, vpr	23	gzip, equake, mgrid, mesa	33	galgel, equake, lucas, gzip
4	vortex1, ammp, equake, galgel	14	bzip, vpr, bzip, equake	24	facerec, applu, fma3d, lucas	34	facerec, gcc, galgel, apsi
5	gcc, galgel, apsi, crafty	15	galgel, crafty, vpr, swim	25	gap, applu, parser, facerec	35	mesa, mcf, swim, sixtrack
6	applu, equake, art, facerec	16	mcf, wupwise, applu, mesa	26	mcf, apsi, twolf, ammp	36	mesa, sixtrack, equake, bzip
7	applu, gap, gcc, parser	17	applu, parser, apsi, perlbmk	27	swim, sixtrack, ammp, applu	37	mcf, gap, gcc, vortex1
8	gap, swim, twolf, mesa	18	mgrid, perlbmk, gzip, mgrid	28	art, fma3d, swim, parser	38	facerec, lucas, mcf, parser
9	sixtrack, fma3d, apsi, vortex1	19	mcf, sixtrack, gcc, apsi	29	apsi, gcc, vortex1, twolf	39	twolf, eon, mesa, lucas
10	ammp, bzip, equake, parser	20	ammp, gcc, art, mesa	30	mgrid, gzip, apsi, equake	40	apsi, gzip, mcf, equake

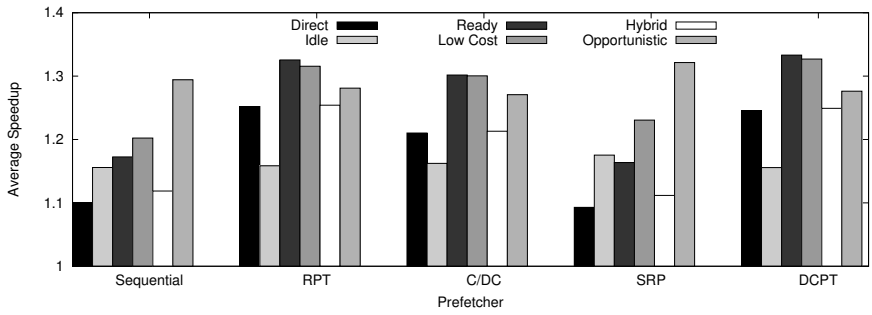


Figure G.2: Average speedup for all cores over all workloads for different scheduling strategies and prefetchers.

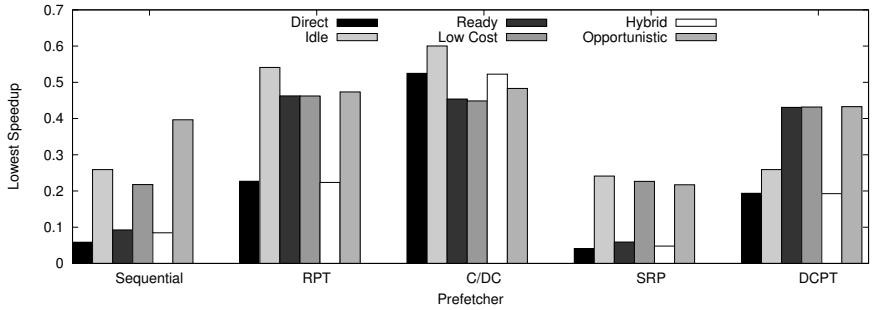


Figure G.3: Lowest speedup for any core in any workload for different scheduling strategies and prefetchers.

prefetchers because it issues less prefetch requests due to it's strict policy. The difference between the *Low cost* and *Ready* strategies is also apparent. In combination with high accuracy prefetchers such as RPT, C/DC and DCPT, the *Ready* strategy performs better than *Low Cost*. In combination with low accuracy prefetchers, the situation is reversed. Finally, the performance of the *Hybrid* strategy is between the *Ready* and *Direct* strategies.

G.5.2 Maximum Performance Regression

Prefetching can drastically increase average system performance. Because prefetching is a speculative technique it might also lead to performance regressions on some workloads. In Figure G.3, we show the lowest speedup for any core on any workload for all the prefetch scheduling strategies. Overall, we observe that the prefetchers with low accuracy shows the largest performance regressions. The strategies utilizing prefetch accuracy measurements (*Low cost*, *Opportunistic*) perform quite well as they are mostly able to adapt to this situation, thus avoiding large performance regressions. The *Direct* strategy shows quite large performance regressions because it does not differentiate between prefetches and demand reads. Thus, a prefetcher which issues many useless prefetches can saturate off-chip bandwidth and delay demand reads. The *Idle* strategy has comparatively low performance regressions, because the strategy is inherently defensive and only issues prefetches when there are no demand reads in the queue. *Hybrid* uses accuracy estimates to select between two strategies (*Direct* and *Ready*). By increasing the threshold, the behaviour of this strategy can be made more similar to the *Ready* strategy.

G.5.3 Accuracy and Coverage

Figure G.4 shows the average accuracy of every workload in each combination of prefetcher and prefetch scheduling strategy. In this figure, we measure the ratio of

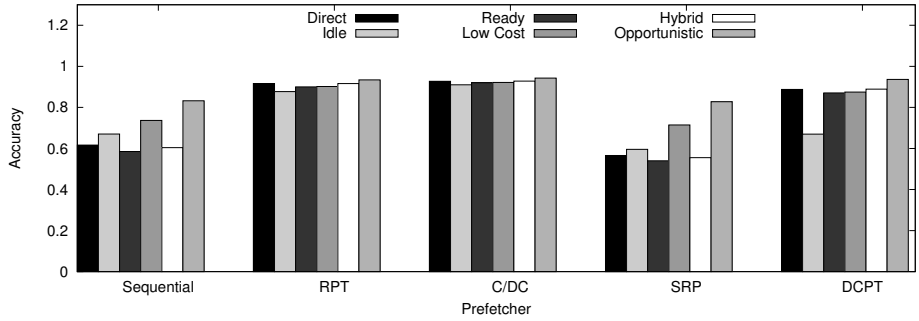


Figure G.4: Average accuracy for all workloads.

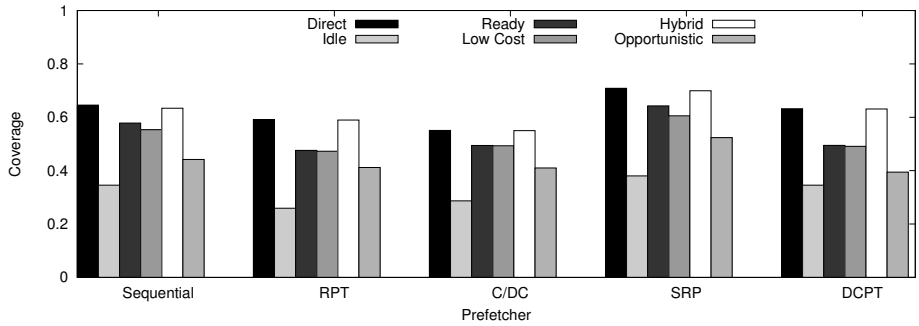


Figure G.5: Average coverage for all workloads.

successful prefetches to the number of issued prefetches by the memory controller for each workload and calculate the arithmetic average of these ratios. This ratio is not necessarily the same as the accuracy of the prefetcher itself, because the prefetch scheduling strategy may drop prefetches. However, the *Direct* strategy does not drop prefetches and issues all prefetches generated by the prefetcher. By examining the results for sequential and SRP, we observe that *Idle*, *Low Cost* and *Opportunistic* are able to achieve higher degrees of accuracy than the *Direct* approach. In the high accuracy prefetchers, there is little difference in the scheduling strategies.

Figure G.5 shows the average coverage of every workload in each combination of prefetcher and prefetch scheduling strategy. Because the *Direct* strategy issues every prefetch generated by the prefetcher, it has a very high coverage. Conversely, *Idle* has very low coverage because it issues few prefetches. *Opportunistic* has a comparatively low coverage compared to the other prefetch scheduling strategies because it issues fewer prefetches than the other strategies. The *Hybrid* strategy is very near the performance of the *Direct* strategy in terms of coverage. This is due to the value of the accuracy threshold used in our experiments. *Low Cost* has

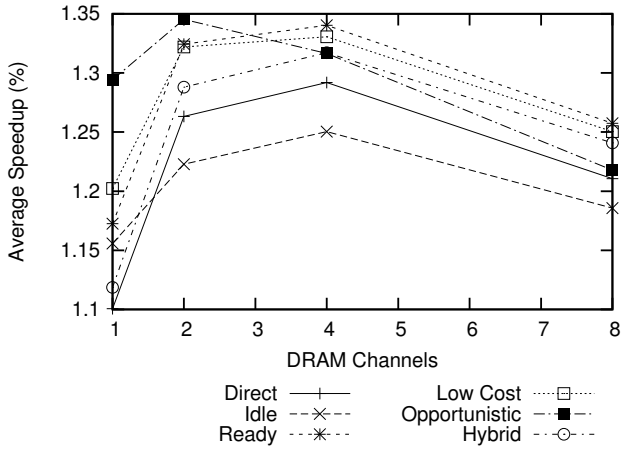


Figure G.6: Effect of increasing the amount of bandwidth available on sequential prefetching.

a slightly lower coverage than *Ready* for the low accuracy prefetchers (sequential, SRP), because it drops prefetches when the accuracy becomes too low.

G.5.4 Increasing DRAM Bandwidth

Figure G.6 and G.7 shows the effect of increasing the amount of bandwidth in the system for sequential and RPT prefetching respectively. Note that the speedup is computed versus a system with the same amount of bandwidth but with no prefetching. Thus, the speedup for 8 DRAM channels is lower than for 1 DRAM channel, although the performance is higher. For sequential prefetching, we observe that the relative performance of *opportunistic* versus the other prefetch scheduling strategies is highest in low bandwidth situations. Furthermore, we observe that *Idle* performs well compared to the other scheduling algorithms with one channel, but relatively worse when more bandwidth is available. Because there is more bandwidth available, the amount of time the memory controllers are idle is increased, leading to more issued prefetches, thus performance is increased. However, *idle* is a very defensive strategy and fails to exploit the additional bandwidth as effectively as the other prefetch scheduling policies.

For RPT prefetching we observe a similar pattern as the amount of bandwidth is increased. RPT is a high accuracy prefetcher and the aggressive strategies such as *Direct*, *Low Cost*, *Hybrid* and *Ready* performs better when bandwidth is increased compared to *Idle* and *Opportunistic*.

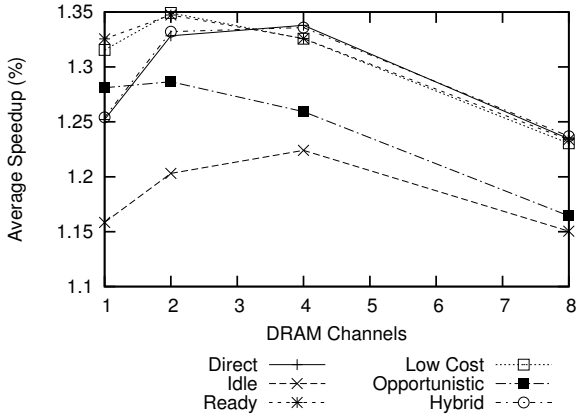


Figure G.7: Effect of increasing the amount of bandwidth available on RPT prefetching.

G.6 Discussion

Three of the strategies (*Low Cost*, *Hybrid* and *Opportunistic*) examined in this paper utilizes prefetch accuracy measurements. All of these use a threshold value to switch between two strategies. Typically, if accuracy is high, then an aggressive strategy is used. Conversely, if accuracy is low, then a more restrictive strategy is used. Thus, the behaviour of these strategies can be adapted by changing the threshold value. In this paper, we have used the same threshold value for all three strategies for easier comparison. The value we have chosen matches the value used in previous work [4, 8].

In this work, we use the same method for estimating prefetch accuracy for all strategies. This method uses special bits in the cache to mark prefetched cache blocks (cache tagging). However, *Opportunistic* prefetch scheduling offers another method. All prefetches generated by the prefetcher are stored in the PVT, while the cache tagging method only stores prefetches that have been issued and completed. Thus, the cache tagging method measures the combined accuracy of the prefetcher and the scheduling strategy, while the PVT combined with the cache tagging method can, in addition, isolate the accuracy of the prefetcher. We have examined this method and found that it offers slightly better performance. However, the threshold value has to be changed, because this method measures something different (issued prefetches accuracy vs. generated prefetches accuracy). Therefore, we have opted to use the same estimation technique for all prefetchers to achieve a fair comparison.

Temporal information is lost when using a page vector table to track issued prefetches. Thus, there is no way to reconstruct the order of which prefetches are to be issued

from the page vector table. However, this problem can be reduced by using a large prefetch distance.

G.7 Conclusion

It is clear from our results that no single prefetch scheduling strategy is suitable for every scenario. The best strategy depends on a variety of factors such as: the prefetcher, the memory controller, the amount of memory bandwidth, the application, design complexity and the amount of acceptable performance regressions. For instance, *Idle* is a good option for minimizing performance regressions. On the other end, *Low Cost* and *Ready* provides the highest average performance. *Opportunistic* provides a trade-off between these two. Because it actively reduces performance regressions, it also provides the highest average performance for sequential and SRP prefetchers.

In this paper, we have presented a novel prefetch scheduling strategy called *Opportunistic*. This strategy emphasises the use of open pages to provide good average performance without large performance regressions. It is particularly suited for systems with relatively low amounts of bandwidth combined with highly aggressive prefetchers. As more cores compete for the same shared off-chip bandwidth, utilizing this limited resource becomes more important. *Opportunistic* prefetch scheduling addresses this problem by utilizing open pages to increase effective bandwidth, while using accuracy estimates to avoid bandwidth saturation. We show that *Opportunistic* prefetch scheduling reduces performance regressions by 6.7X and 5.2X, while improving performance by 17 % and 20 % for sequential and scheduled region prefetching, compared to the direct scheduling strategy.

Bibliography

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[2] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth Prefetching. *SIGPLAN Notices*, 41(11), 2006.

[3] T. Chen and J. Baer. Effective Hardware-Based Data Prefetching for High-performance Processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

[4] M. Grannæs, M. Jahre, and L. Natvig. Low-Cost Open-Page Prefetch Scheduling in Chip Multiprocessors. In *XXVI IEEE International Conference on Computer Design (ICCD)*, 2008.

[5] M. Grannæs, M. Jahre, and L. Natvig. Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables. In *Data Prefetching Championships*, 2009.

[6] I. Hur and C. Lin. Adaptive History-Based Memory Schedulers. In *MICRO 37: Proc. of the 37th An. IEEE/ACM Int. Symp. on Microarch.*, pages 343–354, 2004.

[7] *DDR2 SDRAM Specification.* JEDEC Solid State Tech. Association, May 2006.

[8] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controllers. In *MICRO '08: Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, pages 200–209, 2008.

[9] W. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 301–312, 2001.

[10] W. Lin, S. K. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching. *IEEE Transactions on Computers*, 50(11), 2001.

[11] C. Natarajan, B. Christenson, and F. Briggs. A Study of Performance Impact of Memory Controller Features in Multi-processor Server Environment. In *WMPI '04: Proc. of the 3rd Workshop on Memory Perf. Issues*, pages 80–87, 2004.

[12] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. *IEEE Micro*, 25:90–97, 2005.

[13] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An Adaptive Data Cache Prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 135–145, 2004.

[14] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA '00: Int. Symp. on Comp. Arch.*, pages 128–138, 2000.

[15] J. Shao and B. Davis. A Burst Scheduling Access Reordering Mechanism. In *HPCA '07: Proc. of the 13th Int. Symp. on High-Performance Comp. Arch.*, 2007.

[16] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[17] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. *SIGARCH Computer Architecture News*, 34(2):252–263, 2006.

[18] SPEC. SPEC CPU 2000 Web Page. `http://www.spec.org/cpu2000/`.

[19] Z. Zhu and Z. Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *HPCA '05: Proc. of the 11th Int. Symp. on High-Performance Comp. Arch.*, pages 213–224, 2005.

[20] Z. Zhu, Z. Zhang, and X. Zhang. Fine-Grain Priority Scheduling on Multi-channel Memory Systems. *8th Int. Symp. on High-Performance Comp. Arch.*, pages 107–116, 2002.