The Dissertation Committee for Eiman Ebrahimi
certifies that this is the approved version of the following dissertation:

Fair and High Performance Shared Memory Resource Management

Committee:

Yale N. Patt, Supervisor

Nur A. Touba

Keshav Pingali

Derek Chiou

Onur Mutlu

Fair and High Performance Shared Memory Resource Management

by

Eiman Ebrahimi, B.S.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2011

Dedicated to my loving parents, Ahmad and Batool, and my brother Amir

Acknowledgments

Many people have contributed to this dissertation both technically and motivationally. This acknowledgment is my attempt at expressing my gratitude for their help and support.

First and foremost, I thank my advisor Professor Yale Patt. He has taught me most of what I know today about the fundamentals of computer architecture and how to be a caring teacher. His drive to perform high-impact research rooted in fundamentals has always been inspirational to me. I also thank him for providing the resources and environment within the HPS research group that creates the opportunity to do high-quality research and collaborate with people that are both very smart and great human beings. Finally I would like to thank him for the valuable real-life lessons I have learnt from him over the years which will have impact on more than just my future professional career.

I am very grateful to have had Onur Mutlu as a mentor and the opportunity to collaborate closely with him. I have learnt a great deal about how to perform high-quality research from him. His attention to detail and seemingly never-ending energy for making progress in research have always been exemplary for me. I would also like to thank Onur for his patience with me both technically and personally, and also his always encouraging attitude.

I thank Chang Joo Lee for being a great collaborator and beyond that, a friend. His contributions have without doubt made this dissertation much stronger. He mentored me throughout our collaboration and led by example with his professionalism and discipline. I also thank him for his support and encouragement through the hard times of my graduate student life.

I thank José A. Joao for the motivation and support he provided in getting me started with my PhD research. I am very grateful for the time and effort he

generously spent in the role of a senior colleague helping me grow in the HPS group. I would also like to thank him for his technical feedback and constructive criticisms and suggestions which have made this dissertation stronger. Lastly, his expertise and hard work in maintaining the group's infrastructure has been a significant help to our research group's progress.

I would like to thank Rustam Miftakhutdinov, M. Aater Suleman, Veynu Narasiman, Khubaib, Milad Hashemi, and Faruk Guvenilir for all of their technical feedback and criticism. I thank Rustam for his generous help with the simulation infrastructure work and collaboration on the parallel application memory scheduling work. I also thank Aater for his insightful comments which helped in developing many key ideas, and Milad for proofreading all the chapters of this dissertation.

I thank Francis Tseng, Hyesoon Kim, Moinuddin K. Qureshi, Daniel N. Lynch, Santhosh Srinath and other previous HPS members for their encouragement and friendship. I also thank Leticia Lira for her outstanding administrative support to the HPS group. Besides the HPS group members, I would like to express my gratitude to the following people and organizations.

I thank Keshav Pingali, Nur Touba, and Derek Chiou for serving on my dissertation committee and providing me valuable comments on this dissertation. I also thank Tse-Yu Yeh, Evan Speight, Mootaz Elnozahy and Mark Stephenson for providing me with great experiences as an intern at PA Semi and IBM.

I would like to thank my friends Mahnaz Sadoughi, Hossein Namazi, Kaveh Majlesi, Setareh Nematollahi, Hadi Esmaeilzadeh, and Ali Akhavan for their friendship, encouragement and support during many difficult times of my graduate student life.

I would like to also pay tribute to the memory of the late Professor Margarida Jacome. If it was not for her, I would almost certainly not have come to UT Austin, and would have not had the great experiences and opportunities that came with it. May she rest in peace.

Finally, I cannot express with words my indebtedness to my parents, Ahmad Ebrahimi and Batool Jazbi, and my brother Amir, for their unconditional love, encouragement and support in every step of my life. It is from them that I learnt that hard work, perseverance, and faith in one's abilities are necessary for success in any endeavor. Without them, this work would be meaningless.

Eiman Ebrahimi
December 2011, Austin, TX

Fair and High Performance Shared Memory Resource Management

Eiman Ebrahimi, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Yale N. Patt

Chip multiprocessors (CMPs) commonly share a large portion of memory system resources among different cores. Since memory requests from different threads executing on different cores significantly interfere with one another in these shared resources, the design of the shared memory subsystem is crucial for achieving high performance and fairness.

Inter-thread memory system interference has different implications based on the type of workload running on a CMP. In multi-programmed workloads, different applications can experience significantly different slowdowns. If left uncontrolled, large disparities in slowdowns result in low system performance and make system software's priority-based thread scheduling policies ineffective. In a single multi-threaded application, memory system interference between threads of the same application can slow each thread down significantly. Most importantly, the *critical path* of execution can also be significantly slowed down, resulting in increased application execution time.

This dissertation proposes three mechanisms that address different shortcomings of current shared resource management techniques targeted at multi-programmed workloads, and one mechanism which speeds up a single multi-threaded application by managing main-memory related interference between its different threads.

With multi-programmed workloads, the key idea is that both demand- and prefetch-caused inter-application interference should be taken into account in shared resource management techniques across the entire shared memory system. Our evaluations demonstrate that doing so significantly improves both system performance and fairness compared to the state-of-the-art. When executing a single multi-threaded application on a CMP, the key idea is to take into account the inter-dependence of threads in memory scheduling decisions. Our evaluation shows that doing so significantly reduces the execution time of the multi-threaded application compared to using state-of-the-art memory schedulers designed for multi-programmed workloads.

This dissertation concludes that the performance and fairness of CMPs can be significantly improved by better management of inter-thread interference in the shared memory resources, both for multi-programmed workloads and multi-threaded applications.

Table of Contents

Acknowledgments **v**

Abstract **viii**

List of Tables **xv**

List of Figures **xvii**

Chapter 1. Introduction **1**

1.1 The Problem 1

1.1.1 Inter-Application Interference In Multi-Programmed Workloads 2

1.1.2 Inter-Thread Interference In Multi-Threaded Workloads . . . 4

1.2 Thesis Statement 5

1.3 The Solution: Managing Inter-Thread Memory System Interference for Multi-Core Systems 6

1.3.1 Multi-Programmed Workloads 6

1.3.2 Multi-Threaded Workloads 7

1.4 Contributions 8

1.5 Dissertation Organization 10

Chapter 2. Background and Related Work **11**

2.1 Research in Caching 11

2.2 Research in DRAM Systems 12

2.2.1 Network Fair Queuing (NFQ) 13

2.2.2 Parallelism-Aware Batch Scheduling (PARBS) 14

2.2.3 Thread Cluster Memory Scheduling (TCM) 14

2.2.4 Prefetch-Aware DRAM Controllers (PADC) 15

2.3 Research in Management of Multiple Shared Resources in CMPs . . 15

2.4 Research in Prefetching 18

2.4.1 Per-Core Prefetcher Control 18

2.4.2 Eliminating Useless Prefetches 19

2.4.3 Reducing Cache Pollution 19

2.4.4 Prefetching in shared memory multiprocessors 20
2.5 Other Research in Inter-Thread Interference Management Mechanism 20
2.6 Research in Critical Path Prediction of Parallel Applications 21

Chapter 3. Hierarchical Prefetcher Aggressiveness Control 22
3.1 Introduction 22
3.2 Motivation 26
3.2.1 Shortcomings of Local-Only Prefetcher Control 26
3.3 Hierarchical Prefetcher Aggressiveness Control (HPAC) 28
3.3.1 Local Aggressiveness Control Structure 28
3.3.2 Global Aggressiveness Control Structure 29
3.3.2.1 Terminology 29
3.3.2.2 Global Control Mechanism 31
3.3.2.3 Handling Multiple Prefetchers on Each Core 36
3.3.2.4 Support for System-Level Application Priorities . . . 36
3.3.2.5 Optimizing Threshold Values and Decision Set . . . 37
3.3.3 Implementation 37
3.4 Methodology 39
3.4.1 Metrics 39
3.4.2 Processor Model 40
3.4.3 Workloads 41
3.4.4 Prefetcher Aggressiveness Levels and Thresholds for Evaluation 41
3.5 Experimental Evaluation 43
3.5.1 8-core System Results 43
3.5.2 4-core System Results 47
3.5.2.1 Overall Performance 47
3.5.2.2 Case Study 50
3.5.3 HPAC Performance with Different DRAM Scheduling Policies 54
3.5.4 Effect of HPAC on Fairness 55
3.5.5 HPAC on Systems with Hardware Prefetch Filtering 56
3.5.6 Multiple Types of Prefetchers per Core 57
3.5.7 Sensitivity to System Parameters 58
3.5.8 Hardware Cost 59
3.6 Conclusion 59

Chapter 4. Fairness via Source Throttling **61**
4.1 Introduction . . . 61
4.2 Background and Motivation . . . 63
4.2.1 Shared CMP Memory Systems . . . 63
4.2.2 Motivation . . . 64
4.3 Fairness via Source Throttling . . . 68
4.3.1 Runtime Unfairness Evaluation Overview . . . 68
4.3.2 Dynamic Request Throttling . . . 69
4.3.3 Unfairness Evaluation Component Design . . . 72
4.3.3.1 Cache Interference . . . 73
4.3.3.2 DRAM Bus and Bank Conflict Interference . . . 74
4.3.3.3 DRAM Row-Buffer Interference . . . 75
4.3.3.4 Slowdown Due to Throttling . . . 75
4.3.3.5 Implementation Details . . . 76
4.3.4 System Software Support . . . 76
4.3.5 General Dynamic Request Throttling . . . 78
4.3.6 Hardware Cost and Implementation Details . . . 81
4.3.7 Lightweight FST . . . 81
4.4 Methodology . . . 83
4.4.1 Metrics . . . 83
4.4.2 Processor Model . . . 84
4.4.3 Workloads . . . 84
4.4.4 FST Parameters Used in Evaluation . . . 86
4.5 Experimental Evaluation . . . 86
4.5.1 2-core System Results . . . 87
4.5.2 4-core System Results . . . 90
4.5.2.1 Overall Performance . . . 90
4.5.2.2 Case Study . . . 92
4.5.3 Effect of Throttling Mechanisms . . . 96
4.5.4 Evaluation of System Software Support . . . 98
4.5.5 Effects of Implementation Constraints . . . 100
4.5.6 Effects of Different Sources of Interference . . . 101
4.5.7 Evaluation of Lightweight FST . . . 101
4.5.8 Sensitivity to Unfairness Threshold . . . 102
4.5.9 Effect of Multiple Memory Controllers . . . 103
4.5.10 Evaluation of Using Profile Information . . . 103
4.6 Conclusion . . . 104

Chapter 5. Prefetch-Aware Shared-Resource Management 106
5.1 Introduction . . . 106
5.2 Summary from Previous Chapters and Background . . . 109
5.2.1 Fairness in the Presence of Prefetching . . . 109
5.2.2 Hierarchical Prefetcher Aggressiveness Control (HPAC) . . . 109
5.2.3 Fairness via Source Throttling (FST) . . . 110
5.3 Motivation . . . 111
5.4 High Performance and Fair Shared Resource Management in the Presence of Prefetching . . . 115
5.4.1 Demand Boosting . . . 115
5.4.2 Prefetch-Aware Resource-Based Management Techniques . . . 117
5.4.2.1 Parallelism-Aware Batch Scheduling . . . 117
5.4.2.2 Network Fair Queuing . . . 118
5.4.3 Prefetch-Aware Source-Based Management Techniques . . . 119
5.4.3.1 Determining Application Slowdown in the Presence of Prefetching . . . 120
5.4.3.2 Coordinated Core and Prefetcher Throttling . . . 121
5.5 Methodology . . . 123
5.5.1 Metrics . . . 123
5.5.2 Processor Model . . . 123
5.5.3 Workloads . . . 124
5.5.4 Parameters Used in Evaluation . . . 126
5.6 Experimental Evaluation . . . 126
5.6.1 NFQ Results . . . 126
5.6.2 PARBS Results . . . 129
5.6.2.1 Case Study . . . 130
5.6.3 FST Results . . . 133
5.6.4 Effect on Homogeneous Workloads . . . 135
5.6.5 Sensitivity to System and Algorithm Parameters . . . 135
5.6.6 Hardware Cost . . . 136
5.7 Conclusion . . . 137

Chapter 6. Parallel Application Memory Scheduling 139
6.1 Introduction . . . 139
6.2 Mechanism: Parallel Application Memory Scheduling . . . 141
6.2.1 Runtime System Extensions . . . 142
6.2.1.1 Estimating Limiter Threads . . . 143

6.2.1.2 Measuring Loop Progress . . . 145
6.2.2 Memory controller design . . . 146
6.2.2.1 Terminology . . . 146
6.2.2.2 Prioritization among limiter threads . . . 147
6.2.2.3 Prioritization among non-limiter threads . . . 150
6.2.3 Implementation Details . . . 154
6.3 Methodology . . . 156
6.3.1 Processor Model . . . 156
6.3.2 Benchmarks . . . 156
6.3.3 Parameters Used in Evaluations . . . 157
6.4 Results and Analysis . . . 158
6.4.1 Case Study . . . 160
6.4.2 Comparison to Memory scheduling using Thread Criticality Predictors . . . 164
6.4.3 Sensitivity to System Parameters . . . 165
6.5 Conclusion . . . 165

Chapter 7. Conclusion and Future Research Directions 167
7.1 Conclusion . . . 167
7.2 Future Research Directions . . . 169

Bibliography 171

Vita 178

List of Tables

3.1 Global control rules - ACC_i: Accuracy of prefetcher, BWC_i: Consumed bandwidth, POL_i: Pollution imposed on other cores, and $BWNO_i$: Sum of needed bandwidth of other cores 34

3.2 Baseline system configuration 40

3.3 Characteristics SPEC 2000/2006 benchmarks that appear in evaluated workloads with/without prefetching: IPC, MPKI, Bus Traffic (M cache lines), and ACC 42

3.4 Prefetcher configurations 43

3.5 HPAC threshold values 43

3.6 Summary of average results on the 8-core system 44

3.7 Summary of average results on the 4-core system 47

3.8 Most frequently exercised cases for HPAC in case study I 54

3.9 Stream and GHB with HPAC (local policy: FDP) 58

3.11 Effect of our proposal on Hspeedup (HS) and bus traffic with different system parameters on a 4-core system 59

3.12 Hardware cost of HPAC - Including both local and global throttling structures on an N-core CMP with S_{cache} MB L2 cache 60

4.1 Hardware cost of FST on a 4-core CMP system 82

4.2 Baseline system configuration 84

4.3 Characteristics of 29 SPEC 2000/2006 benchmarks: IPC and MPKI (L2 cache Misses Per 1K Instructions) 85

4.4 FST parameters 86

4.5 Summary of results on the 2-core system 89

4.6 Sensitivity of alone performance to # of MSHRs 98

5.1 Baseline system configuration 124

5.2 Characteristics of 29 SPEC 2000/2006 benchmarks that appear in the workloads of this chapter: IPC and MPKI (L2 cache Misses Per 1K Instructions) with and without prefetching, HPKI (L2 cache Hits Per 1K Instructions) with prefetching, and prefetcher accuracy and coverage 125

5.3 Effect of our proposal on homogeneous workloads in system using NFQ memory scheduling 135

5.4 Effect of our proposal on system using NFQ memory scheduling with different microarchitectural parameters 136

5.5 Hardware cost of our proposed enhancements 137

6.1 Hardware storage cost of PAMS 154
6.2 Baseline system configuration 156
6.3 Benchmark summary 157
6.4 Parameters used in evaluation 157
6.5 Reduction in execution time of PAMS compared to TCP-based [3] memory scheduling 164
6.6 Sensitivity of PAMS performance benefits to memory system parameters 165

List of Figures

1.1 Motivating example 3
1.2 System performance and memory bus traffic with prefetching normalized to no prefetching 4
1.3 Normalized execution time 5

3.3 Speedup of each application w.r.t. when run alone 27
3.4 System performance 27
3.5 Example of how to measure BWC_i, BWN_i, and $BWNO_i$ 31
3.6 HPAC performance on 8-core system (all 32 workloads) 44
3.9 Case Study: individual application behavior 52
3.10 Case Study: system behavior 53
3.14 HPAC performance on 4-core system using HW prefetch filtering (all 32 workloads) 58

4.1 Disparity in slowdowns due to unfairness 62
4.2 Shared CMP Memory System 64
4.3 Access pattern and memory-related stall time of requests when application A running alone (a, b), application B running alone (c, d), A and B running concurrently with no fairness control (e, f), fair cache (g, h), and fair source throttling (i, j) 66
4.4 FST's interval-based estimation and throttling 69
4.5 Changes made to the memory system 83
4.6 Average performance of FST on the 2-core system 87
4.7 Hspeedup of 18 2-core workloads normalized to no fairness control 87
4.8 Average performance of FST on the 4-core system 90
4.9 Normalized speedup of ten 4-core workloads 91
4.10 Unfairness of ten 4-core workloads 92
4.11 Case Study - individual application behavior 93
4.12 Case study - system behavior 93
4.13 Case study - application throttling levels 95
4.14 Effects of different throttling mechanisms for FST 97
4.15 Enforcing thread weights with FST 99
4.16 Enforcing maximum slowdown with FST 99
4.17 Comparing overall results with different system level targets 100

4.18 Effect of periodic updates on FST's performance and unfairness . . 101
4.19 Sensitivity of FST to taking into account different interference sources 102
4.20 Comparing overall results of original and lightweight FST 102
4.21 Sensitivity of FST to unfairness threshold 103
4.22 Effect of FST on a system with two memory controllers 104
4.23 Effect of using profile information for throttling related slowdown . 104

5.1 Harmonic mean of speedups and maximum slowdown on system using NFQ memory scheduler (normalized to FR-FCFS) 108
5.2 Example 1 - Different policies for treatment of prefetches in PARBS batch formation 112
5.3 Memory service timeline for requests of Figure 5.2 113
5.4 Example 2 - No demand boosting vs. Demand boosting 114
5.5 Average system performance and unfairness on 4-core system with NFQ 127
5.6 System performance (Hspeedup) for each of the 15 workloads with NFQ (legend same as Figure 5.5) 128
5.7 Average system performance and unfairness on 4-core system with PARBS 129
5.8 System performance (Hspeedup) for each of the 15 workloads with PARBS(legend same as Figure 5.7) 130
5.9 PARBS case study 132
5.10 Average system performance and unfairness on 4-core system with FST 133
5.11 System performance (Hspeedup) for each of the 15 workloads . . . 134
5.12 Sensitivity to boosting threshold 136

6.1 Normalized execution time 140
6.2 Overview of parallel application memory scheduling 142
6.3 Code-segment based classification 149
6.4 Time based classification 149
6.5 Threads have similar memory behavior 152
6.6 Threads have different memory behavior 152
6.7 Overall Results 159
6.8 Execution of *is* benchmark with different memory scheduling techniques 163

Chapter 1

Introduction

1.1 The Problem

Chip multiprocessor (CMP) systems are generally used to execute two different types of workloads: *Multi-programmed workloads* and *multi-threaded workloads*. In multi-programmed workloads each core of the CMP executes an independent application and there is little to no inter-dependence between the different threads of execution. In a multi-threaded workload, the CMP exploits parallelism by concurrently executing multiple threads of the workload on different cores to speed up a single application.

CMPs are commonly designed such that they share a large portion of memory system resources among different cores (e.g., shared caches, memory controller, etc.). Memory requests from different threads[1] executing on different cores of a CMP interfere significantly with one another with respect to these shared memory resources. This interference is due to both demand memory requests and speculative prefetch requests causing significant delays for memory requests of concurrently executing threads. These delays slow down the execution of each thread compared to the thread executing alone with the entire memory system to itself. From a system design standpoint, the slowdown suffered by different threads of execution has different implications based on the type of workload being executed. In the following subsections we introduce the problems created by *inter-application interference* in multi-programmed workloads, and *inter-thread interference* in parallel multi-threaded workloads.

[1]In multi-programmed workloads each *thread* of execution is an independent application. In multi-threaded workloads multiple interdependent *threads* work together to speed up a single application.

1.1.1 Inter-Application Interference In Multi-Programmed Workloads

We define the slowdown ($ISlowdown$) of thread i as:

$$ISlowdown_i = \frac{T_i^{shared}}{T_i^{alone}}$$

where T_i^{shared} is the number of cycles it takes to run thread i simultaneously with other threads and T_i^{alone} is the number of cycles it would have taken thread i to run alone[2] on the same system. The slowdown experienced by each thread of a workload can be significantly different from the slowdown of the other threads. If left uncontrolled, large disparities in slowdowns can a) result in low system performance and vulnerability to denial of service attacks [51, 73], b) make system-software's priority-based thread scheduling policies ineffective [20] and c) cause highly unpredictable program performance which makes performance analysis and optimization extremely difficult [51, 54, 57].

Figures 1.1 and 1.2 illustrate the problem. In this example four equal-priority applications (each consisting of a single thread) execute one per core on a a 4-core CMP in two configurations: with and without an aggressive prefetcher enabled for each core. Figure 1.1 (a) shows the individual slowdown of each application compared to the application executing alone on the 4-core system. Figure 1.1 (b) shows system unfairness in each configuration. We define system unfairness as:

$$Unfairness = \frac{MAX\{ISlowdown_0, ..., ISlowdown_{N-1}\}}{MIN\{ISlowdown_0, ..., ISlowdown_{N-1}\}}$$

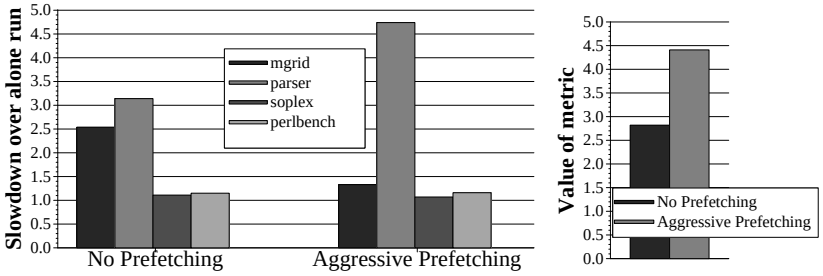
where $ISlowdown_i$ is the slowdown of thread i as defined above.

Two observations from this example illustrate the problem:

1. In the no prefetching case, due to different memory behavior of the applications (different levels of memory intensity, cache behavior, DRAM row buffer

[2]When an application executes alone, the other cores are idle. The running application has the whole memory system to itself.

behavior, etc.), the ratio of the slowdown of the application showing the greatest slowdown that of the application showing the smallest slowdown is almost a factor of 3. The unfairness metric in Figure 1.1(b) indicates exactly this.[3] Unfairness happens when at least one thread slows down more than others as a result of sharing memory system resources. Figure 1.1(a) shows that the slowdown (performance loss) that threads *mgrid* and *parser* suffer as a result of sharing the memory subsystem among the four threads is far more than that which *soplex* and *perlbench* experience. We would like the slowdowns of the applications in a workload to: a) be as close as possible to each other (which would bring the corresponding system unfairness close to the value *one*), and b) each be as close as possible to the value one (which would mean each application executes as fast as it would when executing alone).



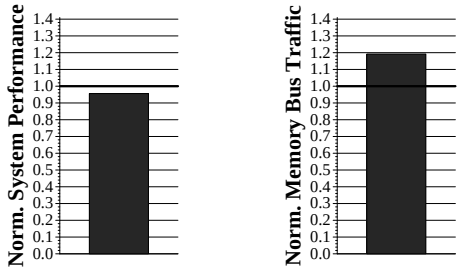
(a) individual application behavior (b) Unfairness

Figure 1.1: Motivating example

2. When prefetching is employed, it has different effects on the slowdowns of the different applications. Some applications are more prefetch friendly than others and benefit more from aggressive prefetching. However, more importantly from a multi-core system perspective, prefetching for each thread will have system-wide effects which alter the slowdowns of concurrently running threads. We refer to these effects as *prefetcher-caused inter-thread (or inter-core) interference*. These effects can cause the disparity between the most slowed down application and the least slowed down application to increase, as is the case in the example shown in

[3]Our system configuration for this experiment is discussed in Section 4.4. The unfairness metric is discussed in Section 3.4.1.

Figure 1.1(a). Figure 1.1(b) shows the corresponding increase in system unfairness. Figure 1.2 (a) shows system performance of the system shown in Figure 1.1 with aggressive prefetching normalized to when no prefetching is used. Figure 1.2 (b) shows the corresponding bus traffic. Figure 1.2 and Figure 1.1(b) show that enabling prefetching in this workload results in lower system performance, higher bus traffic, and higher system unfairness compared to no prefetching. This makes prefetching harmful for this workload even though there are applications in the workload that can significantly benefit from prefetching. The reason for these negative results is unmanaged prefetcher inter-thread interference.



(a) System Performance (b) Bus Traffic

Figure 1.2: System performance and memory bus traffic with prefetching normalized to no prefetching

1.1.2 Inter-Thread Interference In Multi-Threaded Workloads

In parallel multi-threaded workloads, memory requests from threads of the same application interfere with one another in the shared memory subsystem, slowing each thread down significantly. Most importantly, the *critical path* of execution can also be significantly slowed down, resulting in increased application execution time.

To illustrate the importance of DRAM-related inter-thread interference to parallel application performance, Figure 1.3 shows the potential performance improvement that can be obtained for six different parallel applications running on a 16-core system. In this experiment we ideally eliminate all DRAM-related inter-

ference caused by concurrently executing threads of each application.[4] A thread *i*'s DRAM-related interference cycles are those extra cycles that thread *i* has to wait for memory due to bank or row-buffer conflicts caused by concurrently executing threads (compared to if thread *i* were accessing the same memory system alone). In the ideal, unrealizable system we model for this experiment: 1) a thread *i*'s memory requests wait for DRAM banks only if the banks are busy servicing requests from that same thread *i*, and 2) no DRAM row-conflicts occur as a result of some other thread *j* ($i \neq j$) closing a row that is accessed by thread *i*. That is, we model each thread as having its own row buffer in each bank. Figure 1.3 shows that significant performance improvement could potentially be obtained by better management of memory-related inter-thread interference in a parallel application. That is, eliminating inter-thread interference in each application reduces the average execution time of these 6 applications by 45%.

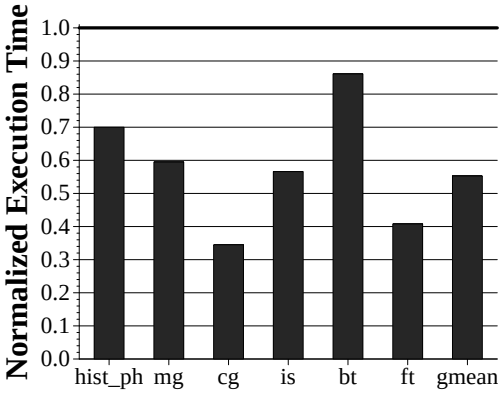


Figure 1.3: Normalized execution time

1.2 Thesis Statement

CMP memory systems can be designed to achieve higher system performance and improved fairness by managing in a coordinated manner, inter-thread interference due to both demand and prefetch requests across *the entire shared memory system*.

[4]Our system configuration and benchmark selection are discussed in Section 6.3.

1.3 The Solution: Managing Inter-Thread Memory System Interference for Multi-Core Systems

The goal of managing inter-thread memory system interference is dependent on the type of workloads being executed. Among multiple different applications, the goal is to design a memory system that provides high *system-performance* and *fairness*. For multi-threaded workloads the goal is to reduce the *execution time* of the parallel application.

1.3.1 Multi-Programmed Workloads

In order to design fair and high performance memory systems for multi-programmed workloads, we propose mechanisms that manage inter-thread interference created by both demand requests and speculative prefetch requests across the entire shared memory system.

This thesis shows that in chip multiprocessor systems, we cannot reap the potential benefits of aggressive prefetching if prefetcher-caused inter-thread interference is left unmanaged. For this purpose we develop a mechanism that controls the aggressiveness of the system's prefetchers in a hierarchical fashion, called Hierarchical Prefetcher Aggressiveness Control (HPAC). HPAC dynamically adjusts the aggressiveness of each prefetcher in two ways: *local* and *global*. Local decisions attempt to maximize each core's performance by taking into account only local feedback information. The global mechanism can override the local decisions by taking into account effects and interactions of different cores' prefetchers when adjusting each one's aggressiveness. Chapter 3 analyzes this mechanism.

In order to achieve system software fairness policies in the presence of multiple shared resources in the memory system of CMPs, this thesis develops a low-cost architectural technique that enables fair sharing of the entire memory system without requiring multiple complicated, specialized, and possibly contradictory fairness techniques for different shared resources. To achieve this goal, we propose a fundamentally new mechanism that gathers dynamic feedback information about

the unfairness in the system, and uses this information to dynamically adapt the rate at which the different cores inject requests into the shared memory subsystem such that system-level fairness objectives are met. Chapter 4 analyzes this *source-based throttling* fairness mechanism.

This thesis also demonstrates that when prefetching is employed in systems using fair shared resource management techniques, system performance/fairness may not improve as expected and can degrade even if prefetcher-caused interference is controlled by throttling prefetchers. To mitigate this effect, this thesis provides mechanisms for management of prefetches in systems using fair shared-resource management based on three fundamental ideas: (1) an application's prefetches should be treated similar to demands only when they are predicted to be useful, (2) treating some applications' prefetches like demands can be unfair to some other memory non-intensive applications; hence, the priority of demands from memory-non intensive applications should be boosted above requests of others, and (3) when using source-based throttling for fairness, prefetcher and core throttling decisions should be coordinated in order to improve system fairness and performance. Chapter 5 analyzes this mechanism.

1.3.2 Multi-Threaded Workloads

This thesis designs a memory scheduler targeted at reducing the execution time of parallel applications by managing inter-thread DRAM interference. The design estimates the critical path using a technique we call limiter thread estimation, and also loop progress measurement [6]. We extend the runtime system with a mechanism to estimate a set of *limiter threads* which is likely to include the thread on the *critical path*. This estimate is based on lock contention, which we quantify as the time threads spend waiting to acquire a particular lock. We use the compiler to enable loop progress measurement in order to estimate the progress of each thread towards a barrier synchronization point within a parallel loop.

The memory controller is build on two key principles: a) it prioritizes threads

that are likely to be on the critical path (which are either limiter threads or threads identified to be falling behind in parallel loops) over others, and b) among a group of limiter threads or non-limiter threads, the memory controller shuffles the priority of threads in a way that reduces the time all threads collectively reach their next synchronization point. Chapter 6 analyzes this memory scheduling technique.

1.4 Contributions

This dissertation makes the following contributions.

- This dissertation shows that in CMPs, uncoordinated, local-only prefetchers can lead to significant system performance degradation compared to no prefetching even though each makes "correct" *local* decisions in an attempt to maximize its core's performance.
- This dissertation proposes a low-cost mechanism to improve the performance and bandwidth-efficiency of prefetching and make it effective in CMPs. The proposed mechanism uses a hierarchical approach to prefetcher aggressiveness control. It optimizes overall system performance with *global control* using inter-core prefetcher interference feedback from the shared memory system, while maximizing prefetcher benefits on each core with *local control* using per-core feedback.
- This dissertation introduces a low-cost, hardware-based and system-software-configurable mechanism to achieve fairness goals specified by system software in the *entire* shared memory system. This mechanism collects dynamic feedback on the unfairness of the system and adjusts request rates of the different cores to achieve the desired fairness/performance balance. By performing *source-based* fairness control, we eliminate the need for complicated *individual resource-based* fairness mechanisms that are implemented independently in each resource and that require coordination.

- This dissertation identifies a new problem in multi-core shared resource management: prefetching can significantly degrade system performance and fairness of multiple state-of-the-art shared resource management techniques. This problem still exists even if state-of-the-art prefetcher throttling techniques are used to dynamically adapt prefetcher aggressiveness.

- This dissertation introduces new general mechanisms for handling prefetches within shared resource management techniques in order to synergistically obtain the benefits of both prefetching and shared resource management in a multi-core system. We apply our mechanisms to three state-of-the-art shared resource management techniques and demonstrate in detail how these techniques should be made aware of prefetching. Comprehensive experimental evaluations show that our proposal significantly improves fairness and performance of these techniques in the presence of prefetching.

- This dissertation proposes a runtime-system mechanism to periodically estimate a set of *limiter threads* which is likely to include the thread on the *critical path* for the purpose of memory request prioritization. We also propose a memory request prioritization mechanism that reduces inter-thread interference among a set of parallel threads which are not contending for locks. This mechanism uses dynamic feedback information about the memory system behavior of the threads in order to reduce the time it takes the threads to collectively reach their synchronization point.

- This dissertation proposes a memory scheduling algorithm that takes into account information about limiter thread estimation, loop-progress measurement, and dynamic thread memory behavior to manage inter-thread memory system interference. We show that by doing so our memory controller design significantly improves the performance of parallel applications compared to a state-of-the-art memory controller designed for multi-programmed workloads.

1.5 Dissertation Organization

This dissertation consists of seven chapters. Chapter 2 provides background information on the prior work related to shared resource management and improving prefetching efficiency that we use to compare our work to. This chapter also discusses other prior work related to the proposals of this dissertation. Chapters 3 through 5 address problems with multi-programmed workloads. Chapter 3 proposes a mechanism to control prefetcher-caused inter-core interference by dynamically adjusting the aggressiveness of multiple cores' prefetchers, in order to enable and improve the benefit of prefetching for multi-core systems. Chapter 4 proposes a new approach to providing fair shared resource management in the *entire shared memory system* that eliminates the need for and complexity of developing fairness mechanisms for each individual resource. Chapter 5 proposes mechanisms that both manage shared resources of a multi-core chip to obtain high-performance and fairness while also exploiting the benefits of prefetching. Chapter 6 deals with parallel multi-threaded applications. We propose a memory scheduling algorithm designed specifically for parallel multi-threaded applications. Chapter 7 contains some concluding remarks and offers suggestions for future work.

Chapter 2

Background and Related Work

This chapter discusses prior studies that are relevant to memory system inter-thread interference management with respect to caches, DRAM systems, prefetching, and the management of multiple shared resources. Of the related work in DRAM systems, this chapter provides additional background on the following previously proposed mechanisms which we build upon, or we use as comparison points in future chapters: Network Fair Queuing (NFQ) [57], Parallelism-Aware Batch Scheduling (PARBS) [55], Thread Cluster Memory Scheduling (TCM) [38], and Prefetch-Aware DRAM Controllers (PADC) [43] (Sections 2.2.1 through 2.2.4). Finally, we discuss research in critical path prediction for parallel applications, as it is relevant to mechanisms proposed in Chapter 6.

2.1 Research in Caching

Prior work in fair caching [31, 36, 28, 32, 58] focus on improving fairness in cache access bandwidth and/or cache capacity sharing. These papers ignore how providing fairness in one shared resource (the shared cache) changes the demand on other shared resources (e.g., the memory controller). This altered demand on other shared resources can create a new source of interference. As a result of the unfair policies of other shared resources the fairness benefits from fair cache capacity sharing can be reduced or even overturned.

Nesbit et. al. [58], proposes virtual private caches (VPC) to provide quality of service from the cache and improve memory system fairness. VPC consists of two major components: the VPC arbiter, and the VPC capacity manager. The VPC arbiter manages the shared cache arrays' access bandwidth using fair queuing

scheduling algorithms. The VPC capacity manager improves fairness by dynamically way-partitioning the cache based on shares allocated by system software. In addition to providing fairness in only one shared resource (the shared cache), we show in Chapter 4 how such partitioning of cache space can result in significant system performance degradation compared to no partitioning at all.

Qureshi and Patt [63] propose utility-based cache partitioning (UCP) for high performance run-time partitioning of shared caches. Such techniques focus on improving performance and not on system fairness. As such, the mechanisms proposed in this thesis are applicable to systems employing techniques like UCP and are orthogonal to them.

Prefetching is already a part of most commercial processors. However, none of the related work mentioned above considers the effect of prefetching on the performance and fairness improvements provided by these techniques. This thesis explores this omission.

2.2 Research in DRAM Systems

Prior work in improving memory system fairness and/or DRAM throughput [57, 54, 55, 37, 38] attempt to improve fairness only in the DRAM controller by modifying the memory scheduling policy. We discuss three of these techniques called: Network Fair Queuing (NFQ) [57], Parallelism-Aware Batch Scheduling (PARBS) [55], and Thread Cluster Memory Scheduling (TCM) [38] in detail in the following subsections.

None of these papers consider interference in a shared cache. The evaluation sections of these papers model only private caches to isolate the effects of interference to the memory controller. Similar to prior work in fair caching, none of the related work in improving fairness in DRAM bandwidth consider the effect of prefetching on the performance and fairness improvements provided by these techniques. We discuss the only work on DRAM scheduling that does address prefetches, Prefetch-Aware DRAM Controllers (PADC) in a following subsection.

All of the prior papers mentioned above focus on multi-programmed workloads and contrary to this thesis (Chapter 6), none consider the inter-dependencies between threads in their prioritization decisions. Ipek et. al. [30], propose using a machine learning technique to design a memory controller that learns to optimize scheduling policies. Their technique observes the system state and estimates the long-term performance impact of different actions. In comparison to the memory scheduler for parallel applications proposed in this thesis (see Chapter 6), this technique requires more complex black-box implementation of re-inforcement learning in hardware. Lin et. al., propose hierarchical memory scheduling for multimedia MPSoCs [47]. This design addresses interference between requests coming from different execution cores of the SoC working on the same application by applying the PAR-BS [55] technique among them. As such, it does not take into account the inter-dependencies of parallel applications that this thesis takes into account to reduce the critical path and only attempts to fairly service the different streams of requests from different cores.

2.2.1 Network Fair Queuing (NFQ)

Nesbit et al. [57] propose network fair queuing (NFQ), a memory scheduling technique based on the concepts of fair network scheduling algorithms. NFQ's goal is to provide quality of service to different concurrently executing applications based on each application's assigned fraction of memory system bandwidth. NFQ's QoS objective is that "a thread *i* that is allocated a fraction *F* of the memory system bandwidth will run no slower than the same thread on a private memory system running at that fraction *F* of the frequency of the shared physical memory system." NFQ determines a *virtual finish time* for every request of each thread. A memory request's virtual finish time is the time it would finish on the thread's virtual private memory system (a memory system running at the fraction *F* of the frequency of the shared memory system). To achieve this objective, memory requests are scheduled *earliest virtual finish time first*. NFQ provides no specification of how prefetches should be treated.

2.2.2 Parallelism-Aware Batch Scheduling (PARBS)

Mutlu and Moscibroda [55] propose parallelism-aware batch scheduling (PARBS), a memory scheduling technique aimed at improving throughput by preserving intra-thread bank parallelism while providing fairness by avoiding starvation of requests from different threads. There are two major steps to the PARBS algorithm: First, PARBS generates batches from a number of outstanding memory requests, and ensures that all requests belonging to the current batch are serviced before the formation of the next batch. This batching technique avoids starvation of different threads and is aimed at improving system fairness. Second, PARBS preserves intra-thread bank-level-parallelism while servicing requests from each application within a batch. This step improves system throughput by reducing each thread's memory related stall time. PARBS does not specify how to handle prefetches in either of these two steps.

2.2.3 Thread Cluster Memory Scheduling (TCM)

Kim et. al. propose thread cluster memory scheduling (TCM), a memory scheduling technique designed to address system throughput and fairness separately with the goal of achieving the best of both for multi-programmed workloads. The algorithm detects and exploits differences in memory access behavior across applications. TCM periodically groups applications into two clusters: *latency-sensitive*, and *bandwidth-sensitive*. This is done once every interval (10M cycles in [38]) based on the applications' memory intensity measured in last level cache misses per thousand instructions (MPKI). The least memory intensive threads are put in the latency-sensitive cluster, and others are placed in the bandwidth-sensitive cluster. To improve system throughput, TCM always prioritizes applications in the latency-sensitive cluster over those in the bandwidth-sensitive cluster. To improve fairness, the priorities of applications in the bandwidth-sensitive cluster are periodically shuffled (every 800 cycles in [38]).

As we show in Chapter 6, a state-of-the-art memory scheduling technique

such as TCM, which is designed for multi-programmed workloads, can improve the performance of parallel multi-threaded workloads compared to standard FR-FCFS (First Ready-First Come First Serve) memory scheduling. However, as this thesis demonstrates, a memory scheduling algorithm targeted at managing DRAM interference specifically for multi-threaded applications can significantly reduce application runtime compared to such state-of-the-art techniques.

2.2.4 Prefetch-Aware DRAM Controllers (PADC)

Lee et. al. [43] propose prefetch-aware DRAM controllers. To our knowledge, this is the only prior work that deals with how prefetches should be dealt with in a shared resource. However, this work targets handling prefetches in a DRAM-throughput-oriented FR-FCFS scheduler that is not designed to provide fairness/QoS. In contrast, Chapter 5 of this thesis is the first work to address how prefetches should be considered in *fair/QoS*-capable memory scheduling techniques that are shown to provide significantly higher performance than throughput-oriented DRAM schedulers. Chapter 5 provides generalized prefetch handling techniques not only for memory scheduling but also for a more general source throttling-based management technique that aims to manage multiple shared resources.

2.3 Research in Management of Multiple Shared Resources in CMPs

Bitirgen et al. [4] propose implementing an artificial neural network that learns each application's performance response to different resource allocations. Their technique searches the space of different resource allocations among co-executing applications to find a partitioning in the shared cache and memory controller that improves performance. In contrast to the shared memory system resource management technique we propose in Chapter 4, this prior work requires that resource-based fairness/partitioning techniques be implemented in each individual resource. In addition, it requires more complex, black-box implementation

of artificial neural networks in hardware.

Nesbit et. al. [59] propose an abstraction of virtual private machines (VPM) for shared resource management. The hardware mechanism they use in this work for the partitioning of shared resources is a combination of virtual private caches (VPC) for cache management [58] and the network fair queuing (NFQ) memory scheduler [57]. VPM [59] mainly focuses on providing performance isolation to concurrently executing applications whereas the goal of this thesis is to achieve high system fairness and performance at the same time.

Herdrich et al. [26] observe that the interference caused by a lower-priority application on a higher-priority application can be reduced using existing clock modulation techniques in CMP systems. However, their proposal does not consider or provide fairness to equal-priority applications. Zhang et al. [74] propose a software-based technique that uses clock modulation and prefetcher on/off control provided by existing hardware platforms to improve fairness in current multi-core systems compared to other software techniques. Neither of these prior papers propose an online algorithm that dynamically controls clock modulation to achieve fairness. In contrast, Chapter 4 of this thesis provides: 1) hardware-based architectural mechanisms that continuously monitor shared memory system unfairness at run-time and 2) an online algorithm that, upon detection of unfairness, throttles interfering applications using two new hardware-based throttling mechanisms (instead of coarse-grained clock modulation) to reduce the interfering applications' request rates.

Jahre and Natvig [33] observe that adjusting the number of available last-level cache MSHRs (Miss Status Holding/information Registers [39] keep track of all requests to a cache until they are serviced) can control the total miss bandwidth available to each thread running on a CMP. However, this prior work does not show how this observation can be used by an online algorithm to dynamically achieve a well-defined fairness or performance goal. In contrast to this prior work, Chapter 4 of this thesis, 1) provides architectural support for achieving different well-defined

system-software fairness objectives while also improving system performance, 2) shows that using complementary throttling mechanisms and preventing bank service denial due to FR-FCFS, as done by our proposed fairness via source throttling (FST, see Chapter 4), provides better fairness/performance than simply adjusting the number of available MSHRs (see Section 4.5.3), 3) shows that our FST approach of throttling sources based on unfairness feedback, provides better system fairness/performance than implementing different fairness mechanisms in each individual shared resource.

Zhuravlev et. al. [76] take a pure software-based scheduling approach to the resource contention problem for multi-core memory systems. This paper proposes to detect which pairs of applications are likely to interfere more with each other and to schedule them for execution on cores that share as small a number of resources as possible. Tang et. al. [69] show the negative impacts of memory subsystem resource sharing on real datacenter applications. They also show that pure software-based intelligent thread-to-core mappings can reduce the amount of memory subsystem interference different applications suffer and improve their performance. The mechanisms we propose in Chapter 4 are orthogonal to those proposed by Zhuravlev et. al. and Tang et. al. as we address the problem of inter-core memory system interference in a finer-grained fashion using a hardware/software cooperative approach:

First, the mix of applications to be scheduled may be such that whatever software schedule is chosen, high inter-core interference will exist among the applications sharing multiple memory system resources. In such cases, pure software-based scheduling approaches can not be as effective. However, the fairness via source throttling (FST) mechanism of Chapter 4 can provide performance and fairness improvements since it throttles applications in a fine-grained manner.

Second, even if inter-core interference can be somewhat reduced using better scheduling, after a number of applications have been scheduled to share some memory system resources, an FST like approach can further improve system fair-

ness and performance by dynamically controlling memory system interference at a finer-granularity.

2.4 Research in Prefetching

To our knowledge, there exists no prior work that directly addresses the problem of inter-application prefetcher interference. This is an important problem as it can significantly degrade or totally destroy the benefits of prefetching in multi-core systems even though prefetch-friendly applications are being executed on a CMP. The related papers in prefetcher control, useless prefetch elimination, and cache pollution reduction which can reduce inter-core prefetcher interference as a side effect of their main goals, are summarized below. We also briefly discuss a number of papers that have studied mitigating the negative effects of prefetching in shared memory multiprocessor systems, e.g. [71].

2.4.1 Per-Core Prefetcher Control

Almost all prefetching algorithms contain a design parameter determining their aggressiveness [35, 2, 34, 11, 56]. For example, in many stream or stride prefetcher designs, *prefetch distance* and *prefetch degree* are two parameters that define how aggressive the prefetcher is [67]. Prefetch distance refers to how far ahead of the demand miss stream the prefetcher can send requests, and prefetch degree determines how many requests the prefetcher issues at once.

In applications where a prefetcher's requests are accurate and timely, a more aggressive prefetcher can achieve higher performance. On the other hand, in applications where prefetching is not useful, aggressive prefetching can lead to large performance degradation due to cache pollution and wasted memory bandwidth, and higher power consumption due to increased off-chip accesses. To reduce these problems, prior studies have proposed dynamically changing the aggressiveness of prefetchers or turning off prefetchers based on their accuracy [13, 56, 67, 18]. For example Feedback-Directed Prefetching (FDP) [67] is a prefetcher throttling

technique that collects feedback local to a single prefetcher (i.e., the prefetcher's accuracy, timeliness, and pollution on the local core's cache) and adjusts its aggressiveness based on its usefulness to reduce the negative effects of prefetching.

All such techniques can significantly degrade performance since they can exacerbate inter-thread interference in shared resources. This is because these techniques use only information *local* to the core the prefetcher resides on and do not have a global view of how each prefetcher's behavior in the CMP system affects overall system performance. In contrast, the hierarchical prefetcher aggressiveness control (HPAC) mechanism we propose in Chapter 3 of this thesis, attempts to maximize each core's performance with prefetching, while also taking into account effects and interactions of different cores' prefetchers when adjusting each one's aggressiveness.

2.4.2 Eliminating Useless Prefetches

Many previous proposals address the problem of useless prefetches by proposing mechanisms to intelligently filter them [52, 8, 46, 75, 53, 43]. Making prefetchers more accurate by eliminating useless prefetches is orthogonal to addressing prefetcher-caused inter-thread contention in the shared memory resources of a CMP system. Chapter 3 shows that managing prefetcher-caused inter-thread interference improves system performance significantly even in a system that already uses prefetch filtering to reduce useless prefetches.

2.4.3 Reducing Cache Pollution

Cache pollution caused by prefetches can be reduced by using separate prefetch buffers [44] instead of inserting prefetched data into caches. However, prior research [67] showed that in order to provide significant performance improvements, the size of the prefetch buffers needs to be large (at least 64KB).

Even though each of the techniques discussed in Sections 2.4.1 through 2.4.3 can make prefetchers more accurate and reduce their generated inter-

ference by reducing the number of their inaccurate requests, none directly identify and address prefetcher-caused inter-application interference. This is an important problem, since even accurate prefetch requests of overly aggressive prefetchers can hamper the performance of prefetching in CMP systems.

2.4.4 Prefetching in shared memory multiprocessors

Prior work on prefetching in multiprocessors [13, 71] study adaptivity and limitations of prefetching in these systems. Dahlgren et al. [13] use prefetch accuracy to decide whether to increase or decrease aggressiveness on a per-processor basis, similar to employing FDP on each core's prefetcher independently. Tullsen and Eggers [71] develop a prefetching heuristic tailored to write-shared data in multi-threaded applications. They apply a restructuring algorithm for shared data to reduce false sharing in multi-threaded applications. In contrast, our goal is to make prefetching effective by controlling prefetch-caused *inter-application* interference. Neither of these prior papers solve the problem this thesis targets and they would be ineffective in reducing prefetcher-caused inter-application interference.

2.5 Other Research in Inter-Thread Interference Management Mechanism

Cheng et. al. [10] propose throttling memory requests generated by threads in streaming parallel applications to reduce memory system interference. Their mechanism is a software-based approach that allows only an analytically-determined threshold number of threads to send out requests to memory at any given time to constrain interference among them. Contrary to the parallel application memory scheduling mechanism proposed in Chapter 6 of this thesis which is not restricted to a particular programming model, their solution requires applications to be written in a gather-compute-scatter style of stream programming. Chen et. al. [9] address inter-thread interference in shared caches as opposed to managing interference at the memory controller and propose a thread scheduling mechanism

aimed at allowing for constructive cache sharing among threads of a parallel application. This prior work is orthogonal to the proposals of this dissertation.

2.6 Research in Critical Path Prediction of Parallel Applications

Cai et. al. [6] propose a mechanism for dynamically detecting critical threads in a parallel region. They use iteration counts of a parallel loop to delay threads that are running ahead to save energy, and to give higher priority to predicted critical threads in the issue queue of an SMT core. In this thesis we use iteration counts of parallel loop regions as a small component of our overall parallel application memory scheduler design as described in Section 6.2.1.2. As such, most of our proposals are orthogonal to this prior work. Other prior techniques exploit the idleness of early-arriving threads at barriers to save power [45, 48], which [6] improves over.

Chapter 3

Hierarchical Prefetcher Aggressiveness Control

3.1 Introduction

Memory latency tolerance mechanisms are critical to improving system performance as DRAM speed continues to lag processor speed. Prefetching is one commonly-employed mechanism that predicts the memory addresses a program will require, and issues memory requests to those addresses before the program needs the data. By doing so, prefetching can hide the latency of a memory access since the processor either does not incur a cache miss for that access or incurs a cache miss that is satisfied earlier because the prefetch request already started the memory access.

In a chip-multiprocessor (CMP) system, cores share memory system resources beyond some level in the memory hierarchy. Bandwidth to main memory and a shared last level cache are two important shared resources in almost all CMP designs. Aggressive prefetching on different cores of a CMP, although very beneficial for memory latency tolerance on many applications when they are run alone, can ultimately lead to 1) significant system performance degradation and bandwidth waste compared to no prefetching, or 2) relatively small system performance improvements with prefetching. This is a result of the following types of prefetcher-caused inter-core interference in shared resources: 1) *prefetch-prefetch interference*: prefetches from one core can delay or displace prefetches from another core by causing contention for memory bandwidth and cache space, and 2) *prefetch-demand interference:* prefetches from one core can either delay demand (load/store) requests from another core or displace the other core's demand-fetched blocks from the shared caches. Our goal in this chapter is to develop a hardware

framework that enables large performance improvements from prefetching in CMPs by significantly reducing prefetcher-caused inter-core interference.

Prefetcher-caused inter-core interference can be somewhat reduced if the prefetcher(s) on each core are individually made more accurate. Previous work [75, 23, 67, 43, 18] proposed techniques to throttle the aggressiveness or increase the accuracy of prefetchers. As a side effect, such techniques can also reduce prefetcher-caused inter-core interference compared to a system that enables aggressive prefetching without any prefetcher control. However, proposed prefetcher throttling techniques [23, 67, 18] only use feedback information *local* to the core the prefetcher resides on. Mechanisms that attempt to reduce the negative effects of aggressive prefetching by filtering useless prefetch requests [43, 75] also operate independently on each core's prefetch requests. Not taking into account feedback information about the amount of prefetcher-caused *inter*-core interference is a major shortcoming of previous techniques. We call this feedback information *global (or system-wide) feedback.*

Why is *global feedback* important? Figure 3.1 compares the performance improvement obtained by independently throttling the prefetcher on each core using state-of-the-art feedback-directed prefetching (FDP) [67] to that obtained by an unrealizable system that, in addition to using FDP, *ideally* eliminates all prefetcher-caused inter-core interference in shared memory resources. To model the ideal system, for each core we eliminate all memory request buffer entry conflicts, memory bank conflicts, row buffer conflicts, and cache pollution caused by another core's prefetcher, but we model all similar interference effects caused by the same core's prefetcher. This experiment was performed for 32 multiprogrammed workloads on a 4-core system and Figure 3.2 shows the results of this experiment for all 32 workloads.[1] Independently throttling each prefetcher using FDP improves performance by only 4%. In contrast, if all prefetcher-caused inter-core interference were ideally eliminated, performance would improve by 56% on average. Hence, significant

[1]These are the same workloads shown in Figure 3.8, which constitute five classes of workloads analyzed in Section 3.5.2.

performance potential exists for techniques that control prefetcher-caused inter-core interference. Moreover, we find that, in some workloads, independently throttling the prefetcher on each core degrades system performance because it blindly increases the aggressiveness of accurate prefetchers. However, using global feedback, coordinated and collective decisions can be made for prefetchers of different cores, leading to significant performance and bandwidth-efficiency improvements, as we show in this chapter.

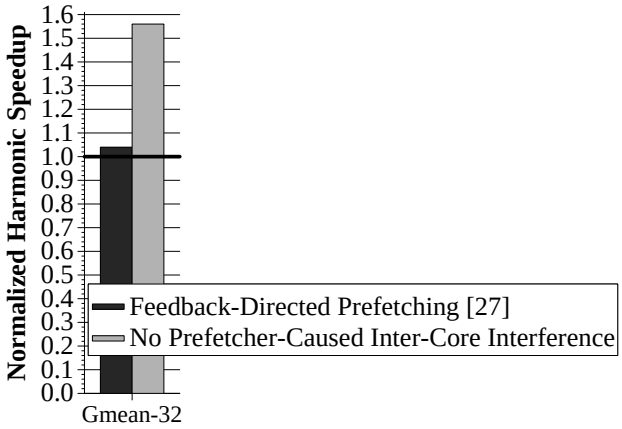
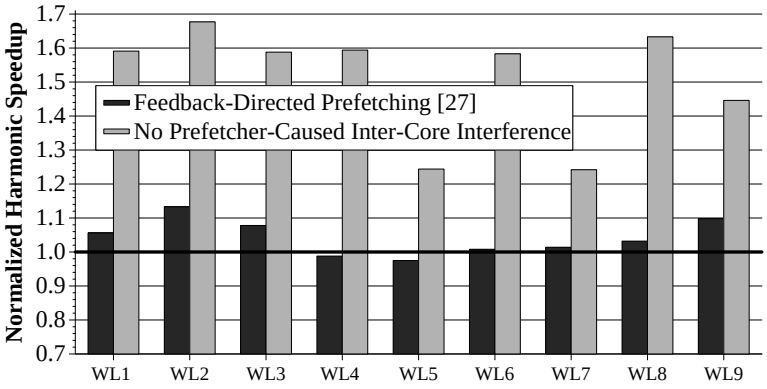
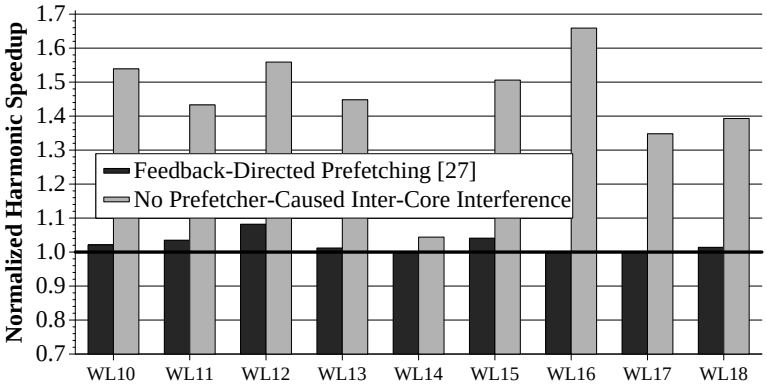


Figure 3.1: Average System performance improvement of ideally eliminating prefetcher-caused inter-core interference vs. feedback-directed prefetching

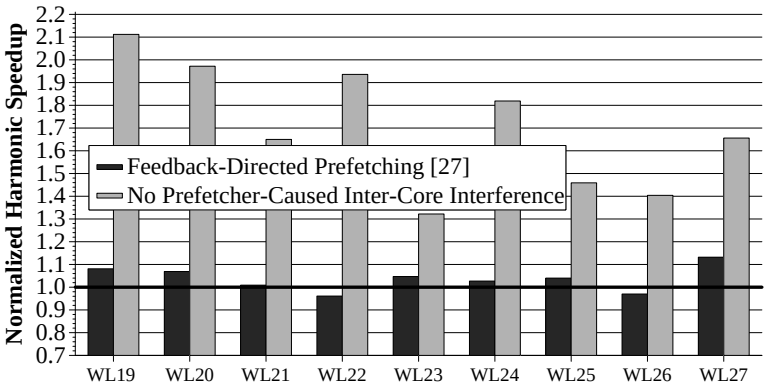
Basic Idea: We develop a mechanism that controls the aggressiveness of the system's prefetchers in a hierarchical fashion, called Hierarchical Prefetcher Aggressiveness Control (HPAC). HPAC dynamically adjusts the aggressiveness of each prefetcher in two ways: *local* and *global*. The local decision attempts to maximize the local core's performance by taking into account only local feedback information, similar to previous prefetcher throttling mechanisms [23, 67, 18]. The global mechanism can override the local decision by taking into account effects and interactions of different cores' prefetchers when adjusting each one's aggressiveness. The key idea is that if prefetcher-caused interference in the shared cache and memory bandwidth is estimated to be significant, the global control system enforces a throttling decision that is best for overall system performance rather than allowing the local control to make a less-informed decision that may degrade over-



(a) Workloads #1-#9 (Class 1)

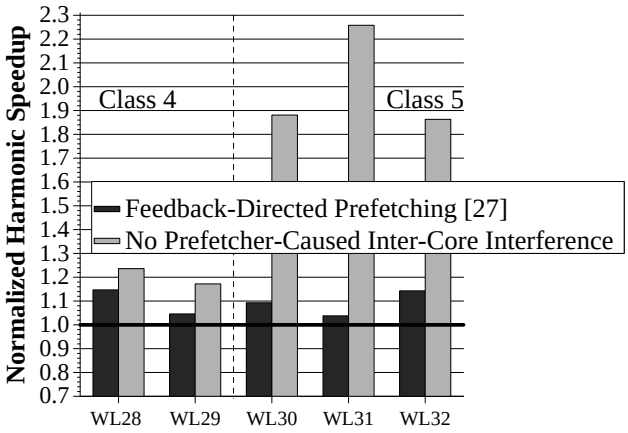


(b) Workloads #10-#18 (Class 2)



(c) Workloads #19-#27 (Class 3)

Figure 3.2: System performance improvement of ideally eliminating prefetcher-caused inter-core interference vs. feedback-directed prefetching (1.0 is the baseline performance with no throttling; performance measured in harmonic speedup, see Section 3.4)



(d) Workloads #28-#32 (Classes 4 and 5)

Figure 3.2: System performance improvement of ideally eliminating prefetcher-caused inter-core interference vs. feedback-directed prefetching (1.0 is the baseline performance with no throttling; performance measured in harmonic speedup, see Section 3.4)

all system performance.

3.2 Motivation

We provided background on relevant previous research in prefetcher aggressiveness control in Section 2.4.1. Since we extensively compare our proposal to Feedback-Directed Prefetching (FDP) [67] in this Chapter, here we describe the shortcomings of this prefetcher control mechanism and more generally *local-only prefetcher control*. We also provide insight into the potential benefits of reducing prefetcher-caused inter-core interference using coordinated control of multiple prefetchers.

3.2.1 Shortcomings of Local-Only Prefetcher Control

Prior approaches to controlling prefetcher aggressiveness that use only information local to each core can make incorrect decisions from a system-wide perspective. Consider the example in Figures 3.3 and 3.4. In the 4-core workload shown, employing aggressive stream prefetching increases the performance of

swim and *lbm* (by 86% and 30%) and significantly degrades the performance of *crafty* and *bzip2* (by 57% and 35%). This results in an overall reduction in system performance of 5% (harmonic speedup - defined in Section 3.4) and an increase in bus traffic of 10% compared to no prefetching. As Figure 3.3 shows, with FDP, applications independently gain some performance, however, even with these gains, system performance still degrades by 4% and bus traffic increases by 7% compared to no prefetching. In contrast, our HPAC proposal makes a coordinated decision for the aggressiveness of multiple prefetchers. As a result, system performance increases by 19.1% (harmonic speedup defined in Section 3.4.1) while bus traffic increases by only 3.5% compared to no prefetching as shown in Figure 3.4.

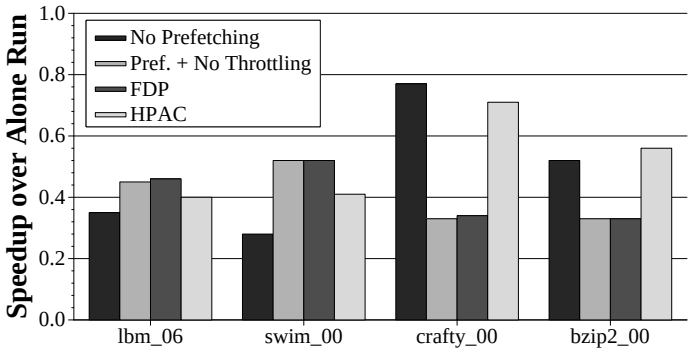
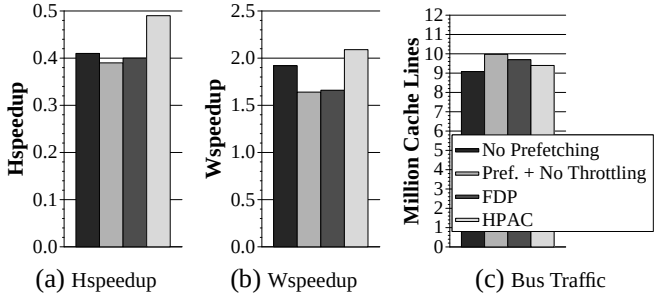


Figure 3.3: Speedup of each application w.r.t. when run alone



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 3.4: System performance

The key to this performance improvement is throttling down of *swim*'s and *lbm*'s prefetchers. When these prefetchers are very aggressive, they cause significant pollution for other applications in the shared cache and cause high contention

for DRAM banks. HPAC detects the interference caused by *swim*'s and *lbm*'s aggressive prefetchers. As a result, even though FDP *incorrectly* decides to throttle up the prefetchers (because the prefetchers are very accurate), HPAC throttles down the prefetchers using global feedback on interference. Doing so results in a loss of *swim*'s and *lbm*'s performance compared to aggressive prefetching. However, this allows *bzip2* to gain performance with prefetching, which was not realizable for this application with no throttling or with FDP, and significantly reduces *crafty*'s performance degradation. Overall, HPAC enables significant performance improvement due to prefetching which cannot be obtained with no throttling or FDP.

The key insight is that a control system that is aware of prefetcher-caused inter-core interference in the shared memory resources can keep an *accurate* but overly aggressive prefetcher in check, whereas a local-only control scheme would allow it to continue to interfere with other cores' memory requests and cause overall system performance degradation.

Our goal: In this chapter, we aim to provide a solution to prefetcher control to significantly improve the performance of prefetching and make it effective on a large variety of workloads in CMP systems. Our HPAC mechanism does exactly that by combining system-wide and per-core feedback information to throttle the aggressiveness of multiple prefetchers of different cores in a coordinated fashion.

3.3 Hierarchical Prefetcher Aggressiveness Control (HPAC)

The Hierarchical Prefetcher Aggressiveness Control (HPAC) mechanism consists of *local* and *global* control structures. The two structures have fundamentally different goals and are hence designed very differently as explained in detail below.

3.3.1 Local Aggressiveness Control Structure

The local control structure adjusts the aggressiveness of the prefetcher(s) of each core with the sole goal of maximizing the performance of that core. This struc-

ture is not aware of the overall system picture and the interference between memory requests of different cores. Prior research [67, 18] proposed such structures. Such previously proposed structures or other novel structures that determine the aggressiveness of a single core's prefetcher(s) are orthogonal to the ideas presented in this chapter and could be incorporated as the local control mechanism of the system proposed here. In fact, we evaluate the use of two previous proposals, FDP [67] and coordinated throttling [18], as our local control structure in Section 3.5.6.

3.3.2 Global Aggressiveness Control Structure

The *global* aggressiveness control structure keeps track of prefetcher-caused inter-core interference in the shared memory system. The global control can accept or override decisions made by each local control structure with the goal of increasing overall system performance and bandwidth efficiency.

3.3.2.1 Terminology

We first provide the terminology we will use to describe the global aggressiveness control. For our analysis we define the following terms, which are used as global feedback metrics in our mechanism:

Accuracy of a Prefetcher for Core i - ACC_i: The fraction of prefetches sent by core i's prefetcher(s) that were used by subsequent demand requests.

Pollution Caused by Core i's prefetcher(s) - POL_i: The number of demand cache lines of all cores j evicted by core i's ($j \neq i$) prefetches that are requested subsequent to eviction.[2] This indicates the amount of disturbance a core's prefetches cause in the cache to the demand-fetched blocks of other cores.

Bandwidth Consumed by Core i - BWC_i: The sum of the number of DRAM banks servicing requests (demand or prefetch) from core i every cycle.

[2] Please note this definition is different from that used by Srinath et al. [67] for pollution caused by inaccurate prefetches on the same core's demands.

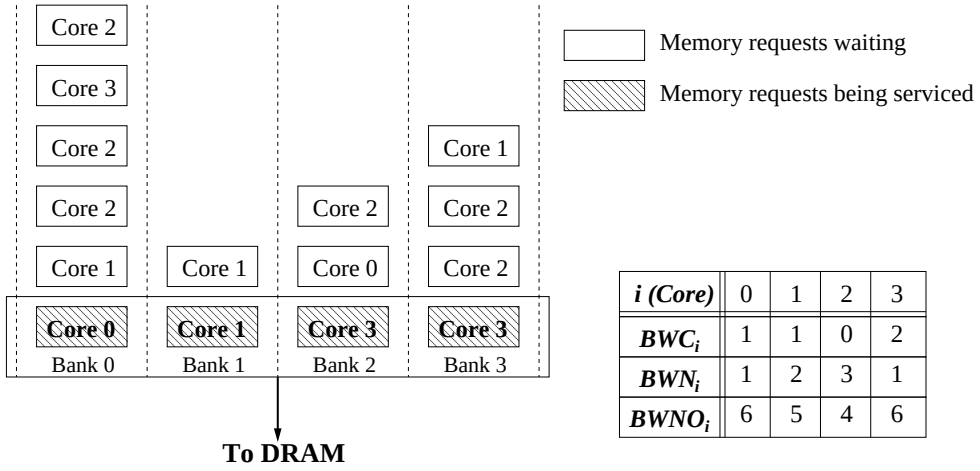
Bandwidth Needed by Core i - BWN_i: The sum of the number of DRAM banks that are busy every cycle servicing requests (demand or prefetch) from cores j when there is a request (demand or prefetch) from core i ($j \neq i$) queued for that bank in that cycle. This indicates the number of outstanding requests of a core that would have been serviced in the DRAM banks had there been no interference from other cores.

Bandwidth Needed by Cores Other than Core i - $BWNO_i$: The sum of the needed bandwidth of all cores except core i for which the prefetcher throttling decision is being made. Therefore,

$$BWNO_i = \sum_{j=0,\ j\neq i}^{N-1} BWN_j, \quad N : Number\ of\ cores$$

Note that the global feedback metrics we define include information on interference affecting *both* demand and prefetch requests of different cores.

Example: Figure 3.5 illustrates the concepts of bandwidth consumption and bandwidth need. Figure 3.5(a) does not show many details of the DRAM subsystem but provides a framework to better understand the definitions above. It shows a snapshot of the DRAM subsystem with four requests being serviced by the different DRAM banks while other requests are queued waiting for those banks to be released. Based on the definitions above, the "Bandwidth consumed by a core" (BWC_i) and "Bandwidth needed by a core" (BWN_i) counts of the four different cores are incremented with the values shown in Figure 3.5(b) in the cycle the snapshot was taken. We focus on the increments for BWN of cores 1 and 2 to point out some subtleties. Core 1 has one request waiting for bank 0, one waiting for bank 1, and one waiting for bank 3. However, when calculating BWN of core 1, only the requests waiting for bank 0 and bank 3 are accounted for. If there was no interference in the system, and if core 1 was the only core using the shared resources, the request from core 1 in the queue for bank 1 would still have had to wait. Hence, the BWN count for core 1 is incremented by 2 in this cycle. Core 2 has three requests waiting for bank 0, one request waiting for bank 2 and two requests waiting for



(a) Snapshot of memory requests' state

i (Core)	0	1	2	3
BWC_i	1	1	0	2
BWN_i	1	2	3	1
$BWNO_i$	6	5	4	6

(b) Measured parameters

Figure 3.5: Example of how to measure BWC_i, BWN_i, and $BWNO_i$

bank 3. However, if there was no interference present only one of the three requests waiting for bank 0, the request waiting for bank 2, and one of the requests waiting for bank 3 could have been serviced in the cycle shown by the snapshot. Hence, the BWN count for core 2 is incremented by 3 in this cycle.

Intuitively, BWC corresponds to the amount of shared bandwidth used by a particular core. A core with high BWC can potentially delay other cores' simultaneous access to the shared DRAM banks and have a negative impact on their memory access latencies. BWN corresponds to the amount of bandwidth a core is *denied* due to interference caused by other cores in the system. A core might be causing interference for other cores if the sum of BWN of other cores grows too large (i.e., $BWNO$ of the core is too large).

3.3.2.2 Global Control Mechanism

In this section we explain how the feedback defined above is used to implement the global control mechanism. We refer to the prefetcher being throttled as the *target* prefetcher. When making a decision to allow or override the decision of a prefetcher's local control, the global control unit needs to know: i) how accurate that prefetcher is, and ii) how much interference the prefetcher is causing for

other cores in the system. In our proposed solution, we use the following parameters to identify how much interference the prefetcher of core *i* is generating for other cores in the shared resources: 1) the bandwidth consumed by core i (BWC_i), 2) the pollution caused by the prefetcher(s) of core *i* on other cores' demand requests (POL_i), and 3) the bandwidth needed by the other cores' requests (both prefetch and demand) ($BWNO_i$). Parameter 1, BWC_i, indicates the potential for increased interference with other cores due to the bandwidth consumption of core i. A high BWC_i indicates that core i will potentially cause interference if the target prefetcher's aggressiveness is not kept in check. Parameters 2 and 3 indicate the existence of such interference in the form of high bandwidth needs of other cores ($BWNO_i$) or cache pollution experienced by other cores (POL_i). When $BWNO_i$ or POL_i has a high value, high interference has been detected, and hence measures are required to reduce it.

Our global control mechanism is an interval-based mechanism that gathers the described feedback parameters during each interval. At the end of an interval, global control uses the collected feedback to allow or override the decision made by the target prefetcher's local control using the following principles.

Principle 1. When the target prefetcher shows low pollution (low POL_i):

(a) If the accuracy of the prefetcher is low[3] and other cores need a lot of bandwidth (i.e., $BWNO$ of the core is high), then override the local control's decision and throttle down. **Rationale:** this state indicates that an inaccurate prefetcher's requests have caused bandwidth interference that is negatively affecting other cores. Hence, the inaccurate prefetcher should be throttled down to reduce the negative impact of its inaccurate prefetches on other cores.

(b) If the accuracy of the prefetcher is low and the prefetcher's core is consuming a large amount of bandwidth (i.e., BWC of the core is high), our global control mechanism allows the local decision to affect the prefetcher only if the local

[3]Note that the local and global control structures can have separate thresholds to categorize an accuracy value as *low* or *high*.

control decides to throttle down. Otherwise, global control leaves the aggressiveness at its current level. **Rationale:** this is a state where interference can potentially worsen because the high bandwidth consumption of an inaccurate prefetcher's core can result in high bandwidth needs for other cores.

(c) If the prefetcher is highly accurate, then allow the local control to decide the aggressiveness of the prefetcher. **Rationale:** if a highly accurate prefetcher is not polluting other cores' demand requests (i.e., POL of the core is low), it should be given the opportunity to increase the performance of its local core.

Principle 2. When the target prefetcher is polluting other cores (high POL_i):

(a) If the accuracy of the prefetcher is low, then override the local control's decision and throttle down. **Rationale:** if an inaccurate prefetcher's requests pollute the demands of other cores, it could be negatively affecting system performance.

(b) If the target prefetcher is highly accurate, then allow the local decision to proceed if there are no other signs of interference (both BWC and $BWNO$ of the core are low). **Rationale:** if the bandwidth needs of all cores are observed to be low, the high pollution caused by the target prefetcher is likely not affecting the performance of other cores.

(c) If either bandwidth consumed (BWC) by the target prefetcher's core is high or other cores need a lot of bandwidth ($BWNO$ is high), then only allow the local decision to affect the prefetcher if it decides to throttle down, otherwise leave aggressiveness at its current level. **Rationale:** even though the prefetcher is accurate, it is showing more than one sign of interference which could be damaging overall system performance.

Rules used for global aggressiveness control: Table 3.1 shows the rules used by the global control structure. There is one case in this table that does not follow the general principles described above, case 14. In this case, interference is quite severe even though the target prefetcher is highly accurate. The target prefetcher's core is consuming a lot of bandwidth and is polluting other cores' demands while other cores have high bandwidth needs. Due to high interference

detected by multiple feedback parameters, reducing prefetcher aggressiveness is desirable. The decision based on general principles would be: "Allow local decision only if it throttles down," which is not strong enough to alleviate this very high interference scenario. Therefore, we treat case 14 as an exception to the aforementioned principles and enforce a throttle-down with global control.

Case	**Info from core *i***			**Info from other cores**	**Decision**	**Rationale**
	Acc_i	BWC_i	POL_i	$BWNO_i$		
1	Low	Low	Low	Low	Allow local decision	No interference
2	Low	High	Low	Low	Allow local throttle down	1(b)
3	Low	-	Low	High	Global enforces throttle down	1(a)
4	High	Low	Low	Low	Allow local decision	1(c)
5	High	High	Low	Low	Allow local decision	1(c)
6	High	Low	Low	High	Allow local decision	1(c)
7	High	High	Low	High	Allow local decision	1(c)
8	Low	Low	High	Low	Global enforces throttle down	2(a)
9	Low	High	High	Low	Global enforces throttle down	2(a)
10	Low	-	High	High	Global enforces throttle down	2(a)
11	High	Low	High	Low	Allow local decision	2(b)
12	High	High	High	Low	Allow local throttle down	2(c)
13	High	Low	High	High	Allow local throttle down	2(c)
14	High	High	High	High	Global enforces throttle down	Very high interference

Table 3.1: Global control rules - ACC_i: Accuracy of prefetcher, BWC_i: Consumed bandwidth, POL_i: Pollution imposed on other cores, and $BWNO_i$: Sum of needed bandwidth of other cores

Classification of global control rules: We group the cases of Table 3.1 into three main categories classified based on the intensity of the interference detected by each case.

1) *Severe interference scenario:* Cases 3, 8, 9, 10 and 14 fall into this category. In these cases, the goal of the global control is to reduce the detected severe interference by reducing the number of prefetch requests generated by the interfering prefetchers. When the target prefetcher is inaccurate, and there is high bandwidth need from other cores (case 3), or when an inaccurate prefetcher is polluting (cases 8, 9 and 10), or when a prefetcher consumes high bandwidth, is polluting, and causes high bandwidth needs on other cores (case 14), prefetcher aggressiveness should be reduced regardless of the local decision. After the prefetcher has been throttled down and the detected interference has become less severe (by either improved accuracy of the target prefetcher, reduced pollution for other cores, or reduced bandwidth needs of other cores), the global throttling decisions for this prefetcher will be relaxed. This will allow the prefetcher to either not be throttled down further or throttled up based on local control's future evaluation of the prefetcher's behavior.

2) *Borderline interference scenario:* Cases 2, 12 and 13 fall into this category. In these cases, the global control's goal is to prevent the prefetcher from transitioning into a severe interference scenario. This is done by either overriding local control throttle up decisions, or throttling the prefetcher down at the request of the local control. When an inaccurate prefetcher consumes high bandwidth but is not polluting (case 2), or when an accurate polluting prefetcher either consumes high bandwidth or causes high bandwidth need for other cores (cases 12 and 13), the prefetcher should not be throttled up as a result of the local control structure's decision.

3) *No interference scenario or moderate interference by an accurate prefetcher:* All other cases fall in this category. In these cases, either there is no interference or an accurate prefetcher has moderate interference. As explained in the general principles, in these cases, the prefetchers' aggressiveness is decided by the local control structures optimizing for highest performance in each core. We empirically found that high prefetcher accuracy can overcome the negative effects

of moderate interference (cases 5, 6, 7 and 11) and therefore the local decision is used.

In Section 3.5.2.2, we present a detailed case study to provide insight into how prefetcher-caused inter-core interference hampers system performance and how HPAC improves performance significantly by reducing such interference.

3.3.2.3 Handling Multiple Prefetchers on Each Core

HPAC can seamlessly support systems with multiple types of prefetchers per core. In such systems, where speculative requests from different prefetchers can potentially increase prefetcher-caused inter-core interference, having a mechanism that takes such interference into account is even more important. In a system with multiple prefetchers on each core, the system-level feedback information referred to in Table 3.1 for each core corresponds to *all* the prefetchers on that core as a whole. For example, accuracy is the overall accuracy of all prefetchers on that core. Similarly, pollution is the overall shared cache pollution caused by all prefetchers from that core.

Note that prior research on intra-core prefetcher management [18] is orthogonal to the focus of this chapter. In HPAC, when the local aggressiveness control corresponding to each core makes a decision for one of the prefetchers on that core, the global control allows or overrides that decision based on the effects and interactions of other cores' prefetchers.

3.3.2.4 Support for System-Level Application Priorities

So far, we have assumed concurrently running applications are of equal priority and hence are treated equally. However, system software (operating system or virtual machine monitor) may make policy decisions prioritizing certain applications over others in a multi-programmed workload. We seamlessly extend HPAC to support such priorities: 1) separate threshold values can be used for each concurrently-running application, 2) these separate threshold values are config-

urable by the system software using privileged instructions. To prioritize a more important application within HPAC, the system software can simply set a higher threshold value for $BWNO_i$, POL_i, and BWC_i and a lower threshold value for Acc_i for that application. By doing so, HPAC allows a more important application's prefetcher to cause more interference for other applications if doing so improves the more important application's performance.

3.3.2.5 Optimizing Threshold Values and Decision Set

Genetic algorithms [25] can be used to optimize the threshold value set or decision set of HPAC at design time. We implemented and evaluated a genetic algorithm for this purpose. We found that the improvements obtained by optimizing the decision set were not significant, but a 5% average performance improvement on top of HPAC can be achieved by optimizing thresholds for subsets of workloads. Although we did not use such an optimization for the results presented in the evaluation section, this demonstrates a rigorous and automated approach for optimizing HPAC's decision and threshold sets.

3.3.3 Implementation

In our implementation of HPAC, FDP, and coordinated throttling, all mechanisms are implemented using an interval-based sampling mechanism similar to that used in [67, 18]. To detect the end of an interval, a hardware counter is used to keep track of the number of cache lines evicted from the L2 cache. When the counter exceeds the empirically determined threshold of 8192 evicted lines, an interval ends and the counters gathering feedback information are updated using the following equation:

$$Count = 1/2\, CountAtStartOfInt. \; + \; 1/2\, CountDuringInt.$$

HPAC's global control mechanism maintains counters for keeping track of the BWC_i, BWN_i and POL_i at each core i as defined in Section 6.2.2.1. ACC_i

is calculated by maintaining two counters to keep track of the number of useful prefetches for core i ($used\text{-}total_i$) and the total number of prefetches of that core ($pref\text{-}total_i$). The update of these counters is similar to that proposed for FDP. ACC_i is obtained by taking the ratio of $used\text{-}total_i$ to $pref\text{-}total_i$ at the end of every interval. BWC_i and BWN_i are maintained by simply incrementing their values at the memory controller every DRAM cycle based on the state of the requests in that cycle (see the example in Section 6.2.2.1).

To calculate POL_i, we need to track the number of last-level cache demand misses core i's prefetches cause for all other cores. We use a Bloom filter [5] for each core i to approximate this count. Each filter entry consists of a *pollution bit* and a *processor id*. When a prefetch from core i replaces another core j's demand line, core i's filter is accessed using the evicted line's address, the corresponding pollution bit is set in the filter, and the corresponding processor id entry is set to j. When memory finishes servicing a prefetch request from core j, the Bloom filters of all cores are accessed by the address of the fetched line and the pollution bit of that entry is reset if the processor id of the corresponding entry is equal to j. When a demand request from core j misses the last level cache, the filters of all cores are accessed using the address of that demand request. If the corresponding bit of core i's Bloom filter is set and the processor id of the entry is equal to j, the filter predicts that this line was evicted previously due to a prefetch from core i and the miss could have been avoided had the prefetch that evicted the requested line not been inserted into the cache. Hence, POL_i is incremented and the pollution bit is reset. The interval-based nature of our technique puts the communication of information needed to update pollution filters and feedback counters off the critical path since all such communication only needs to complete before the end of the current interval.

3.4 Methodology

3.4.1 Metrics

To measure CMP system performance, we use *Individual Speedup (IS)*, *Harmonic mean of Speedups (Hspeedup or HS)* [49], and *Weighted Speedup (Wspeedup or WS)* [66]. Recent research [19] on system-level performance metrics for multi-programmed workloads shows that *HS* is the reciprocal of the *average turn-around time* and is the primary user-oriented system performance metric [19]. *WS* is equivalent to *system throughput* which accounts for the number of programs completed per unit of time. We show both metrics throughout our evaluation. $I_{Speedup}$ is the ratio of an application's performance when it is run together with other applications on different cores of a CMP to its performance when it runs alone on one core in the CMP system (other cores are idle). This metric reflects the change in performance of an application that results from running concurrently with other applications in the CMP system.

To demonstrate that the performance gains of our techniques are not due to unfair treatment of applications, we also report *Unfairness*, as defined in [54]. We use the following definitions in determining unfairness:

1) We define a memory system design as *fair* if the slowdowns of equal-priority applications running simultaneously on the cores of a CMP are the same, similarly to previous works [66, 49, 7, 22, 54].

2) We define slowdown as T_{shared}/T_{alone} where T_{shared} is the number of cycles it takes to run simultaneously with other applications and T_{alone} is the number of cycles it would have taken the application to run alone on the same system.

Unfairness is defined as the ratio between the maximum individual slowdown and minimum individual slowdown among all co-executed applications.

The equations below provide the definitions of these metrics. In these equations, N is the number of cores in the CMP system. IPC^{alone} is the IPC measured when an application runs alone on one core in the CMP system with the prefetcher

enabled (other cores are idle). $IPC^{together}$ is the IPC measured when an application runs on one core while other applications are running on the other cores.

$$HS = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}, \quad WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$$

$$I_{Speedup_i} = \frac{IPC_i^{together}}{IPC_i^{alone}}$$

$$IS_i = \frac{T_i^{shared}}{T_i^{alone}}, \ Unfairness = \frac{MAX\{IS_0, IS_1, ..., IS_{N-1}\}}{MIN\{IS_0, IS_1, ..., IS_{N-1}\}}$$

3.4.2 Processor Model

We use a cycle accurate x86 CMP simulator for our evaluation. We faithfully model all port contention, queuing effects, bank conflicts, and other DDR3 DRAM system constraints in the memory subsystem. Table 3.2 shows the baseline configuration of each core and the shared resource configuration for the 4 and 8-core CMP systems we use.

Execution Core	15 stage out of order processor Decode/retire up to 4 instructions Issue/execute up to 8 micro instructions; 256-entry reorder buffer;
Front End	Fetch up to 2 branches; 4K-entry BTB; 64K-entry hybrid branch predictor
On-chip Caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line size; L1 D-cache: 32KB, 4-way, 2-cycle, 64B line size; Shared unified L2: 2MB (4MB for 8-core), 16-way (32-way for 8-core), 16-bank, 15-cycle (20-cycle for 8-core), 1 port, 64B line size;
Prefetcher	Stream prefetcher with 32 streams, prefetch degree of 4, and prefetch distance of 64 [70, 67]
DRAM controller	On-chip, demand-first [43] Parallelism-Aware Batch Scheduling policy [55] 128 L2 MSHR (256 for 8-core) and memory request buffer; Two memory channels for 8-core;
DRAM and Bus	667MHz DRAM bus cycle, Double Data Rate (DDR3 1333MHz) [50], 8B-wide data bus, 8 DRAM banks, 16KB row buffer per bank Latency: 15ns per command (^tRP, ^tRCD, CL);

Table 3.2: Baseline system configuration

3.4.3 Workloads

We use the SPEC CPU 2000/2006 benchmarks for our experimental evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [62] as a representative portion of each benchmark.

We classify benchmarks into *memory intensive/non-intensive*, *with/without cache locality in data accesses*, and *prefetch sensitive* for purposes of analysis in our evaluation. We refer to a benchmark as memory intensive if its L2 Cache Miss per 1K Instructions (MPKI) is greater than one. We say a benchmark has cache locality if its number of L2 cache hits per 1K instructions is greater than five, and we say it is prefetch sensitive if the performance delta obtained with an aggressive prefetcher is greater than 10% compared to no prefetching. These classifications are based on measurements made when each benchmark was run alone on the 4-core system. We show the characteristics of the benchmarks that appear in the evaluated workloads in Table 3.3.

We used 32 four-application and 32 eight-application multi-programmed workloads for our 4-core and 8-core evaluations. These workloads were randomly selected from all possible 4-core and 8-core workloads with the one condition that the evaluated workloads be relevant to the proposed techniques: each application in each workload is either memory intensive, prefetch sensitive, or has cache locality.

3.4.4 Prefetcher Aggressiveness Levels and Thresholds for Evaluation

Table 3.4 shows the values we use for determining the aggressiveness of the stream prefetcher in our evaluations. The aggressiveness of the GHB [56] prefetcher is determined by its *prefetch degree*. We use five values for GHB's prefetch degree (2, 4, 8, 12, 16). Throttling a prefetcher up/down corresponds to increasing/decreasing its aggressiveness by one level.

Threshold values for FDP [67] and coordinated throttling [18] are empiri-

	No prefetcher			With Stream Prefetcher			
Benchmark	IPC	MPKI	Traffic	IPC	MPKI	Traffic	ACC (%)
bzip2_00	1.27	0.39	0.08	1.37	0.11	0.09	96
swim_00	0.36	23.10	4.62	0.75	3.43	4.62	99.9
facerec_00	1.35	2.72	0.54	1.45	1.18	0.88	59.6
parser_00	1.06	0.62	0.12	1.17	0.09	0.15	86.1
apsi_00	1.75	0.85	0.17	1.87	0.39	0.17	99.3
perlbmk_00	1.85	0.04	0.01	1.86	0.02	0.02	28.7
xalancbmk_06	0.95	0.82	0.16	0.79	1.44	0.85	8.2
libquantum_06	0.39	13.51	2.70	0.40	2.62	2.70	99.9
omnetpp_06	0.41	8.60	1.72	0.44	8.39	5.31	11.5
GemsFDTD_06	0.46	15.35	3.07	0.74	1.67	3.34	90.9
lbm_06	0.37	20.16	4.03	0.50	3.76	4.25	93.9
bwaves_06	0.58	18.7	3.74	1.02	0.57	3.74	99.8
crafty_00	1.89	0.09	0.02	1.92	0.05	0.03	48.2
leslie3d_06	0.37	20.75	4.15	0.63	1.45	4.46	92.7
sphinx3_06	0.36	12.57	2.51	0.59	1.71	4.25	56.7
zeusmp_06	0.74	4.37	0.87	0.85	1.68	1.19	63.5
mesa_00	1.62	0.59	0.12	1.61	0.29	0.12	97.4
gromacs_06	1.06	0.26	0.05	1.07	0.03	0.06	88.2
lucas_00	0.34	10.61	2.12	0.56	0.31	2.12	99.9
equake_00	0.40	19.33	3.87	0.61	3.11	3.93	98.2
vortex_00	1.14	0.90	0.18	1.11	0.90	0.30	20.4
gobmk_06	1.18	0.28	0.06	1.20	0.17	0.08	49.8
eon_00	2.21	0.01	0.00	2.21	0.00	0.00	37.3
soplex_06	0.33	20.93	4.19	0.49	4.69	5.05	79.8
gzip_00	1.15	0.34	0.07	1.15	0.31	0.19	4.3
applu_00	0.67	11.39	2.28	0.95	1.1	2.34	96.9
wrf_06	0.66	7.81	1.56	0.93	0.69	1.64	95
povray_06	1.92	0.02	0.00	1.92	0.01	0.01	27.3
mcf_00	0.16	33.82	6.76	0.10	28.54	24.42	74.9
mgrid_00	0.53	6.49	1.30	0.79	0.38	1.38	94.3
sixtrack_00	0.95	0.10	0.02	0.95	0.00	0.12	46.8
sjeng_06	1.61	0.37	0.07	1.61	0.37	0.12	2.1
fma3d_00	0.84	4.13	0.83	1.27	0.43	0.86	95.1
gap_00	0.90	1.98	0.40	1.44	0.04	0.40	99.1
hmmer_06	1.33	1.11	0.22	1.66	0.01	0.23	96.2
twolf_00	1.19	0.09	0.02	1.20	0.04	0.02	93.4
vpr_00	1.31	0.10	0.02	1.33	0.06	0.02	76.8
apsi_00	1.75	0.85	0.17	1.87	0.39	0.17	99.3
wupwise_00	1.47	1.68	0/34	1.89	0.38	0.61	48.5

Table 3.3: Characteristics SPEC 2000/2006 benchmarks that appear in evaluated workloads with/without prefetching: IPC, MPKI, Bus Traffic (M cache lines), and ACC

cally determined for our system configuration. We use the threshold values shown in Table 3.5 for HPAC. We determined these threshold values empirically, but due to the large design space, we did not tune the values. Unless otherwise stated, we use FDP as the local control mechanism in our evaluations.

Aggressiveness Level	Stream Prefetcher *Distance*	Stream Prefetcher *Degree*
Very Conservative	4	1
Conservative	8	1
Moderate	16	2
Aggressive	32	4
Very Aggressive	64	4

Table 3.4: Prefetcher configurations

ACC	BWC	POL	$BWNO$
0.6	50k	90	75k

Table 3.5: HPAC threshold values

3.5 Experimental Evaluation

We evaluate HPAC on both 4-core and 8-core systems. We find the improvements provided by our technique increase as the number of cores in a CMP increases. We present both sets of results, but to ease understanding most of the analysis is done on the 4-core system.

3.5.1 8-core System Results

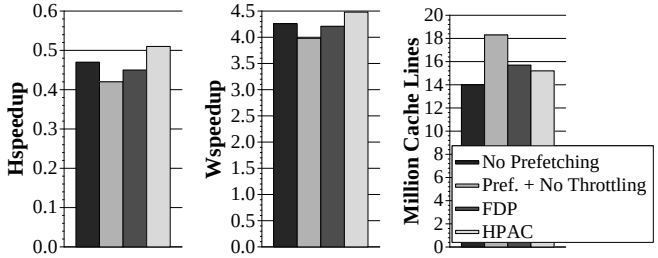
Figure 3.6 shows system performance and bus traffic averaged across 32 workloads evaluated on the 8-core system. HPAC provides the highest system performance among all examined techniques, and is the only technique employing prefetching that improves average system performance over no prefetching. It also consumes the least bus traffic among schemes that employ prefetching. Several key observations are in order:

1. Employing aggressive prefetching with no throttling performs worse than no prefetching at all: harmonic speedup and weighted speedup decrease by 16% and 10% respectively. We conclude that attempting to aggressively prefetch in CMPs with no throttling has significant negative effects, which makes aggressive prefetch-

ing a challenge in CMP systems.

2. FDP increases performance compared to no prefetcher throttling, but is still inferior to no prefetching. FDP's performance is 4.8%/1.2% (HS/WS) lower than no prefetching while its bus traffic is 12.8% higher. We conclude that inter-core prefetcher interference, which is left unmanaged by even a state-of-the-art local-only prefetch control scheme, can cause prefetching to be detrimental to system performance in CMPs.

3. HPAC improves performance by 8.5%/5.3% (HS/WS) compared to no prefetching, at the cost of only 8.9% higher bus traffic. In addition, HPAC increases performance by 23% and 14% (HS), and consumes 17% and 3.2% less memory bandwidth compared to no throttling and FDP respectively, as summarized in Table 3.6. HPAC enables prefetching to become effective in CMPs by controlling and reducing prefetcher-caused interference. Among the schemes where prefetching is enabled, HPAC is the most bandwidth-efficient. We conclude that with HPAC, prefetching can significantly improve system performance of CMP systems without large increases in bus traffic.

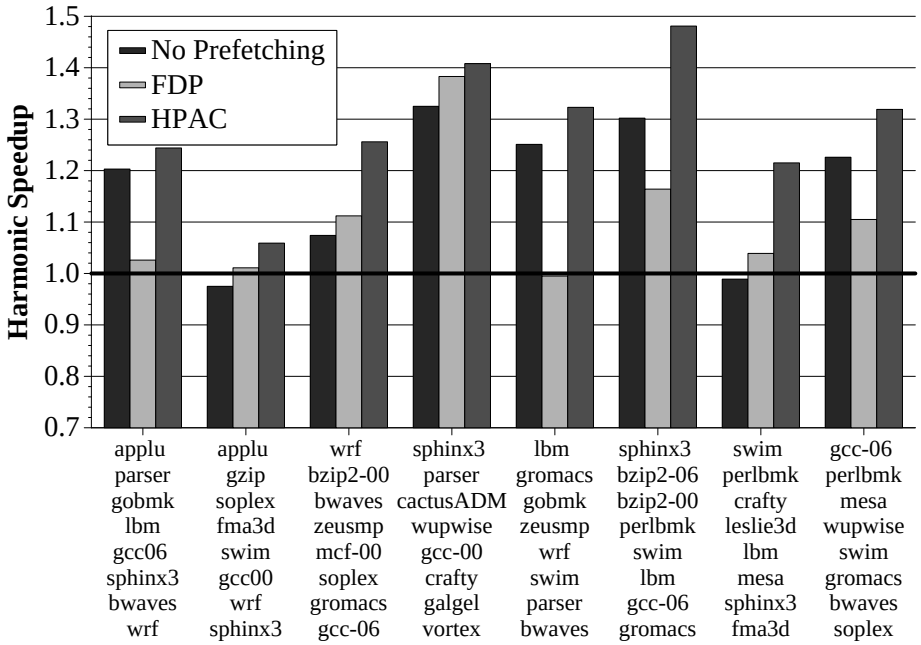


(a) Hspeedup (b) Wspeedup (c) Bus Traffic

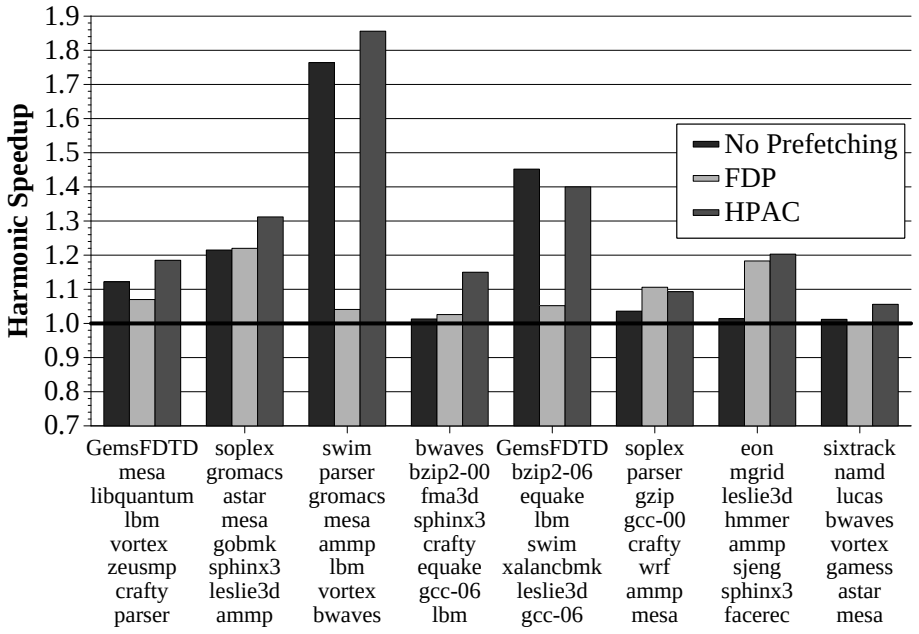
Figure 3.6: HPAC performance on 8-core system (all 32 workloads)

	HS	WS	Bus Traffic
HPAC Δ over No Prefetching	8.5%	5.3%	8.9%
HPAC Δ over No Throttling	23%	12.8%	-17%
HPAC Δ over FDP	14%	6.6%	-3.2%

Table 3.6: Summary of average results on the 8-core system

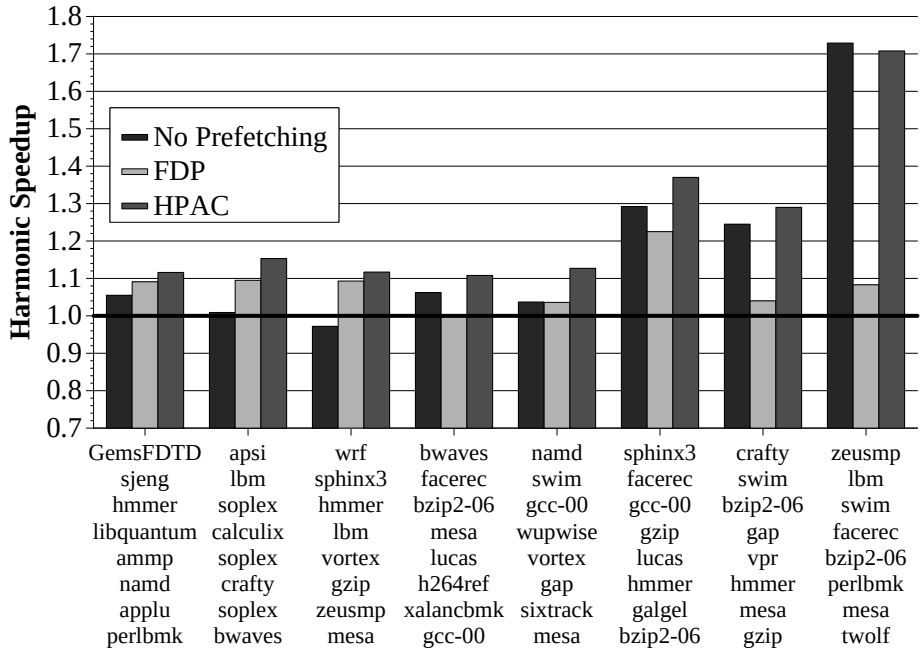


(a) Workloads #1-#8

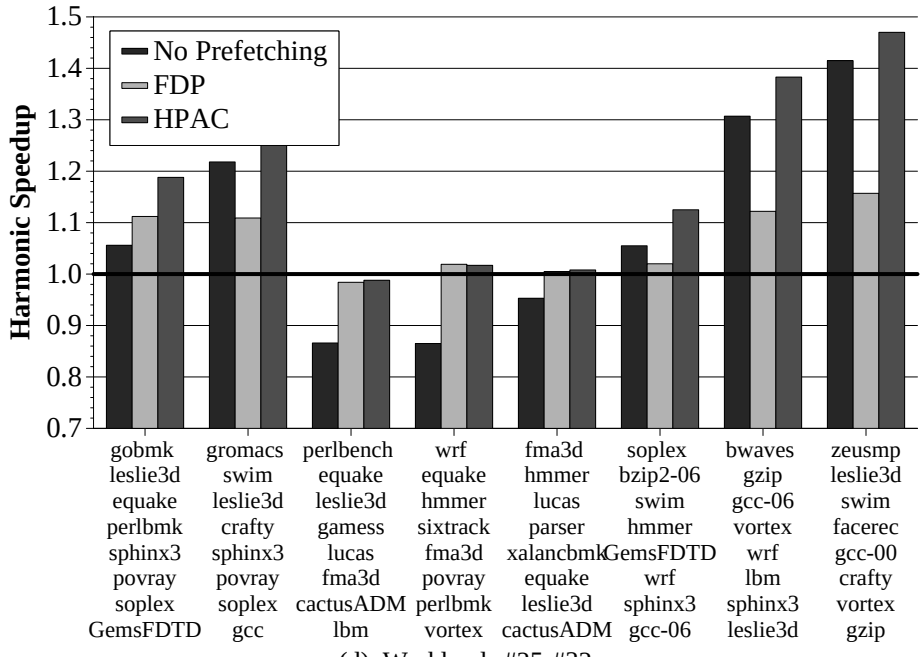


(b) Workloads #9-#16

Figure 3.7: Hspeedup of 8-core workloads
(normalized to “no throttling”)



(c) Workloads #17-#24



(d) Workloads #25-#32

Figure 3.7: Hspeedup of 8-core workloads
(normalized to “no throttling”)

To show how HPAC performs compared to other schemes on different workloads, Figure 3.7 shows the performance improvement (in terms of harmonic speedup) of no prefetching, FDP, and HPAC normalized to that of prefetching with no throttling across the 32 evaluated workloads.

3.5.2 4-core System Results

We first present overall performance results for the 32 workloads evaluated on the 4-core system, and analyze the workloads' characteristics. We then discuss a case study in detail to provide insight into the behavior of the scheme.

3.5.2.1 Overall Performance

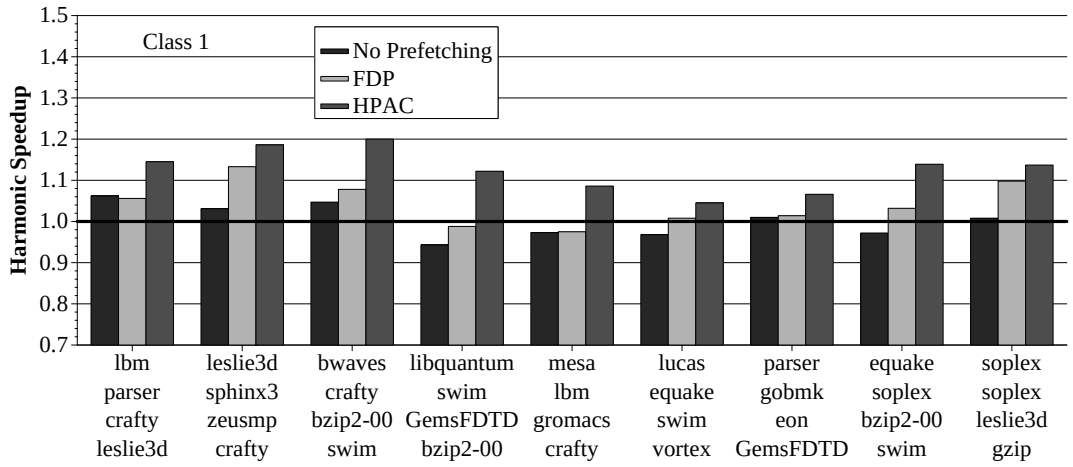
Table 3.7 summarizes our overall performance results for the 4-core system. As observed with the 8-core workloads in Section 3.5.1, HPAC provides the highest system performance among all examined techniques. It also generates the least bus traffic among schemes that employ prefetching.

	HS	WS	Bus Traffic
HPAC Δ over No Prefetching	8.9%	5.3%	8.9%
HPAC Δ over No Throttling	15%	8.4%	-14%
HPAC Δ over FDP	10.7%	4.7%	-3.2%

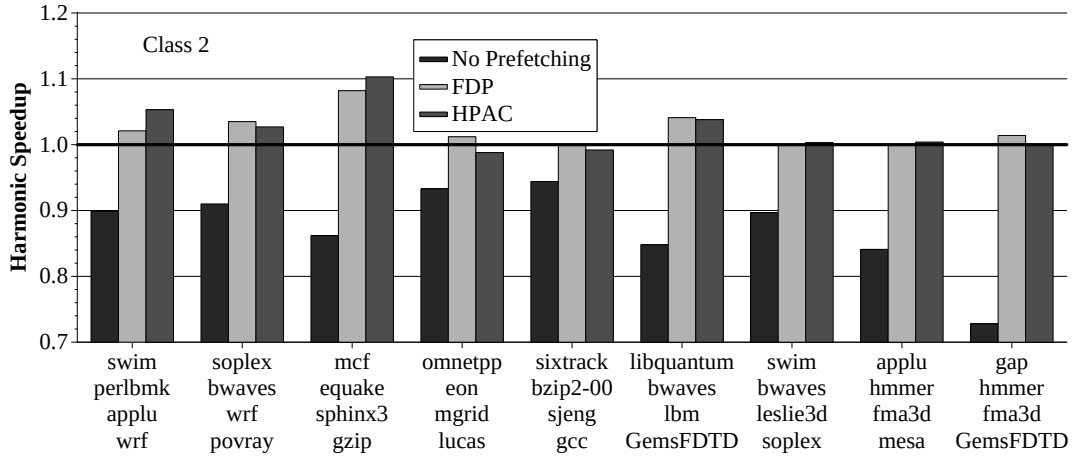
Table 3.7: Summary of average results on the 4-core system

Workload Analysis: Figure 3.8 shows the performance improvement (in terms of harmonic speedup) of no prefetching, FDP, and HPAC normalized to that of prefetching with no throttling across the 32 evaluated workloads. We identify five distinct classes of workloads as shown in subfigures 3.8 (a) through (d).

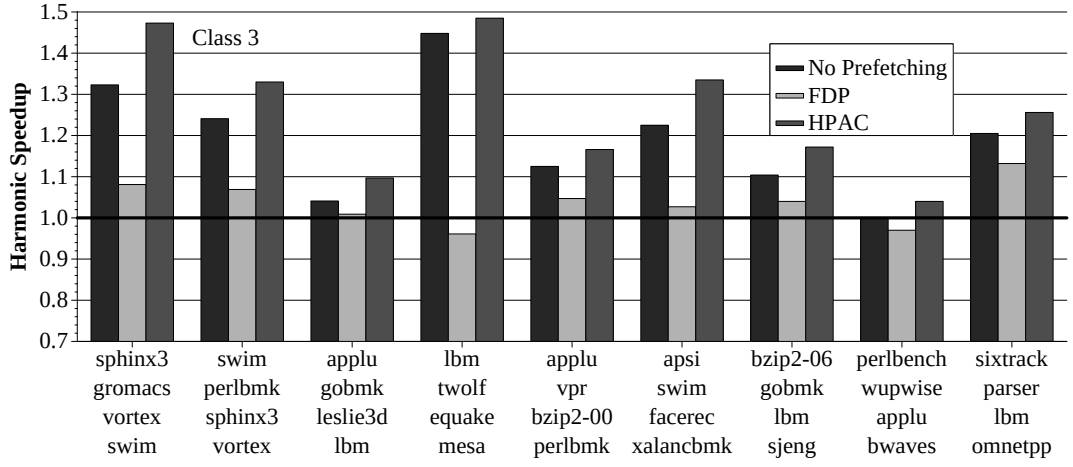
Class 1: Prefetcher-caused inter-core interference does not allow significant gains with no throttling or FDP. In fact, in the leftmost two cases, FDP degrades performance slightly compared to no throttling because it increases prefetchers' interference in the shared resources (as discussed in detail in the case study presented in Section 3.5.2.2). HPAC controls this interference and enables much higher system



(a) Class 1 workloads

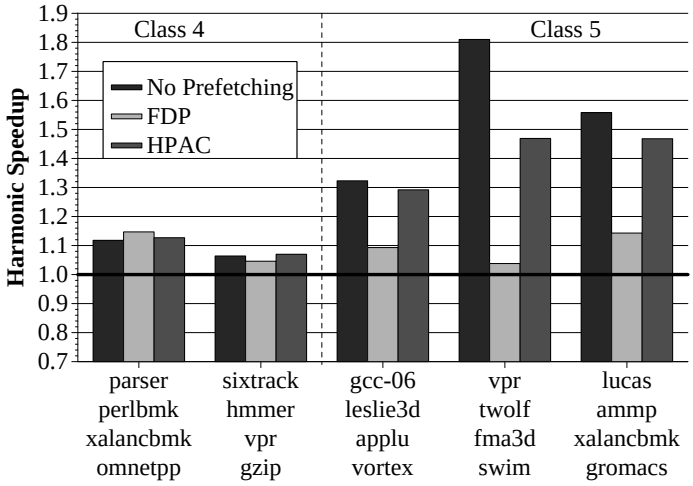


(b) Class 2 workloads



(c) Class 3 workloads

Figure 3.8: Hspeedup of 4-core workload classes
(normalized to “no throttling”)



(d) Classes 4 and 5 workloads

Figure 3.8: Hspeedup of 4-core workload classes (normalized to "no throttling")

performance improvement than what is possible without it.

Class 2: Significant performance can be obtained with FDP and sometimes with no throttling since prefetcher-caused inter-core interference is tolerable. HPAC performs practically at least as well as these previous mechanisms.

Class 3: Intense prefetcher-caused inter-core interference makes prefetching significantly harmful with no throttling or FDP. FDP can slightly reduce this interference compared to no throttling by making prefetchers independently more accurate, but still degrades performance significantly compared to no prefetching. The existence of such workloads makes prefetching without control of prefetcher-caused inter-core interference very unattractive in CMPs. However, HPAC enables prefetching to significantly improve performance over no prefetching.

Class 4: Small prefetcher-caused inter-core interference can be controlled by FDP. Potential system performance to be gained by prefetching is small compared to other classes. Small performance degradations of no throttling can be eliminated using FDP or HPAC, which perform similarly.

Class 5: Intense prefetcher-caused inter-core interference exists due to the co-execution of prefetch-friendly benchmarks together with cache-sensitive and mem-

ory non-intensive applications. FDP can slightly reduce this interference compared to no throttling by making prefetchers independently more accurate, but still degrades performance significantly compared to no prefetching. HPAC detects inter-core interference and throttles down aggressive prefetchers. However it performs worse than no prefetching on these workloads. This is due to unfair treatment of demand requests from cache-sensitive and memory non-intensive applications in the presence of the large number of prefetch requests from the prefetch-friendly and memory intensive applications. We address this problem in detail in Chapter 5.

We conclude that HPAC is effective for a wide variety of workloads. In many workloads where there is significant prefetcher-caused inter-core interference (classes 1 and 3), HPAC is the only technique that enables prefetching to improve performance significantly over no prefetching. When prefetcher-caused inter-core interference is not significant (class 2), HPAC retains significant performance over no prefetching. Hence, HPAC makes prefetching effective and robust in multi-core systems.

3.5.2.2 Case Study

This case study is an example of a scenario where prefetcher-caused inter-core interference that hampers system performance can be observed in both shared bandwidth and shared cache space. It provides insight into why controlling the aggressiveness of a CMP's prefetchers based on local-only feedback from each core is ineffective.

We examine a scenario where a combination of three memory-intensive applications (*libquantum, swim, GemsFDTD*) are run together with one memory non-intensive application that has high data cache locality (*bzip2*). Figures 3.9 and 3.10 show individual benchmark performance and overall system performance, respectively. Several observations are in order:

First, employing aggressive prefetching on all cores improves performance by 6.0%/3.7% (HS/WS) compared to no prefetching. However, the ef-

fect of prefetching on individual benchmarks is mixed: even though two applications' (*swim* and *GemsFDTD*) performance significantly improves, that of two others (*libquantum* and *bzip2*) significantly degrades. Although *libquantum*'s prefetches are very accurate, they, along with *libquantum*'s demands, are delayed by *swim*'s and *GemsFDTD*'s prefetches in the memory controller. Since previous works [43, 18] analyzed the effects of enabling prefetching in multi-core systems, we focus our analysis on the differences between prefetching without throttling, local-only throttling, and HPAC.

Second, using FDP to reduce the negative effects of prefetching actually degrades system performance by 1.2%/1% (HS/WS) compared to no throttling. To provide insight, Figure 3.9(b) shows the percentage of total execution time each application's prefetcher spends in different aggressiveness levels. With FDP, since the feedback indicates high accuracy for prefetchers of *libquantum*, *swim* and *GemsFDTD* (respectively at accuracies of 99.9%, 99.9%, 92%), their prefetchers are kept very aggressive. This causes significant memory bandwidth interference between these three applications, which causes *libquantum*'s demand and prefetch requests to be delayed by the aggressive *swim* and *GemsFDTD* prefetch requests. On the other hand, *bzip2*'s demand-fetched cache blocks get thrashed due to the very large number of *swim*'s and *GemsFDTD*'s prefetches: *bzip2*'s L2 demand MPKI increases by 26% from 2.1 to 2.7. *bzip2*'s prefetcher performance is also affected negatively as its useful prefetches are evicted from the cache before being used and therefore reduced by 40%. This prompts FDP to reduce the aggressiveness of *bzip2*'s prefetcher as a result of detected *local* low accuracy, which in turn causes a loss of potential performance improvement for *bzip2* from prefetching. As a result, FDP does not help *libquantum*'s performance and degrades *bzip2*'s performance, resulting in overall system performance degradation compared to no throttling.

Third, using HPAC increases system performance significantly by 12.2%/8.7% (HS/WS) while reducing bus traffic by 3.5% compared to no throttling. Hence, HPAC makes aggressive prefetching significantly beneficial to the entire

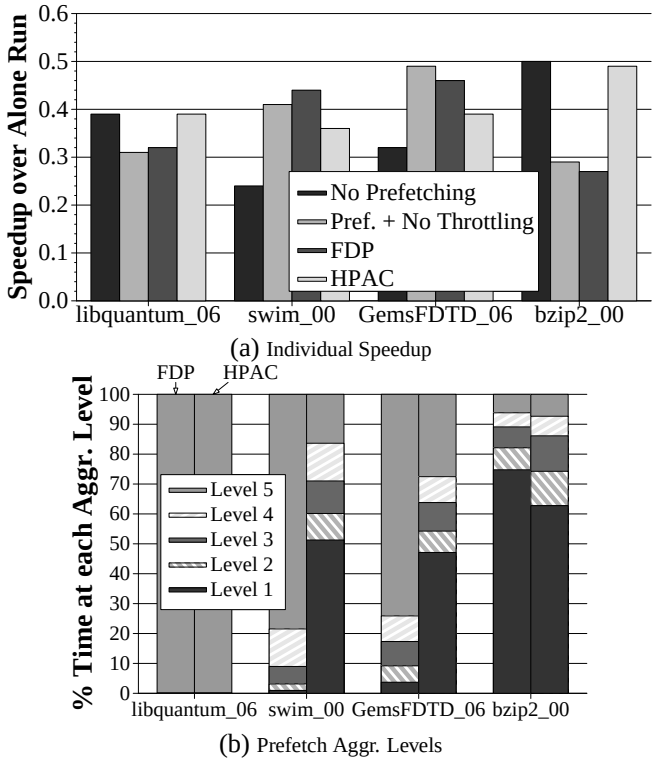
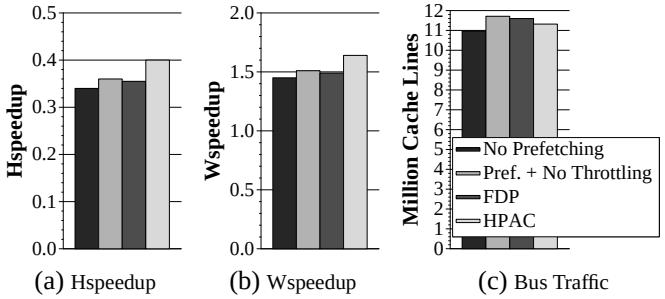


Figure 3.9: Case Study: individual application behavior

system: it increases performance by 19%/12.7% (HS/WS) compared to a system with no prefetching. The main reason for the performance benefits of HPAC over FDP is twofold: 1) by tracking prefetcher-caused interference in the shared cache, HPAC recognizes that aggressive (yet accurate) prefetches of *swim* and *GemsFDTD* destroy the cache locality of *bzip2* and throttles those applications' prefetchers, thereby significantly improving *bzip2*'s locality and performance, 2) by tracking the bandwidth need and bandwidth consumption of cores in the DRAM system, HPAC recognizes that *swim*'s and *GemsFDTD*'s aggressive prefetches delay service of *libquantum*'s demands and prefetches, and therefore throttles down these two prefetchers. Doing so significantly improves *libquantum*'s performance. HPAC improves the performance of all applications compared to no prefetching, except for *bzip2*, which still incurs a slight (1.5%) performance loss. Finally, HPAC reduces memory bus traffic compared to both FDP and no throttling because: 1) it eliminates many unnecessary demand requests that need to be re-fetched from memory



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 3.10: Case Study: system behavior

by reducing the pollution *bzip2* experiences in the shared cache: *bzip2's* bandwidth demand reduces by 33% with HPAC compared to FDP, 2) it eliminates some useless (or marginally useful) prefetch requests due to *GemsFDTD's* very aggressive prefetcher: we found that in total, HPAC reduces the number of useless prefetch requests by 14.6% compared to FDP.

Table 3.8 and Figure 3.9(b) provide more insight into the behavior and benefits of HPAC by showing the most common global control cases (from Table 3.1) for each application and the percentage of time each prefetcher spends at different levels of aggressiveness respectively (in Figure 3.9 (b), Level 1 corresponds to a "very conservative" aggressiveness level as defined in Section 3.4.4.). Note that Case 14, which indicates extreme prefetcher interference is *swim*'s and *GemsFDTD*'s most frequent case. As a result, HPAC throttles down their prefetchers to reduce the interference they cause in shared resources. Figure 3.9(b) shows that FDP keeps these two applications' prefetchers at the highest aggressiveness for more than 70% of their execution time, which degrades system performance, because FDP cannot detect the inter-core interference caused by the two prefetchers. In contrast, with HPAC, the two prefetchers spend approximately 50% of their execution time in the lowest aggressiveness level, thereby reducing inter-core interference and improving system performance.

We conclude that HPAC can effectively control and reduce the shared resource interference caused by the prefetchers of multiple memory and prefetch-intensive applications both among themselves and against a simultaneously running

Application	Most Frequent Case #	2nd Most Frequent Case #	3rd Most Frequent Case #
libquantum	Case 6 (89%)	Case 13 (7%)	Case 7 (2%)
swim	Case 14 (65%)	Case 7 (23%)	Case 6 (6%)
GemsFDTD	Case 14 (55%)	Case 7 (24%)	Case 6 (8%)
bzip2	Case 10 (39%)	Case 3 (39%)	Case 6 (15%)

Table 3.8: Most frequently exercised cases for HPAC in case study I

memory non-intensive application, thereby resulting in significantly higher system performance than what is possible without it.

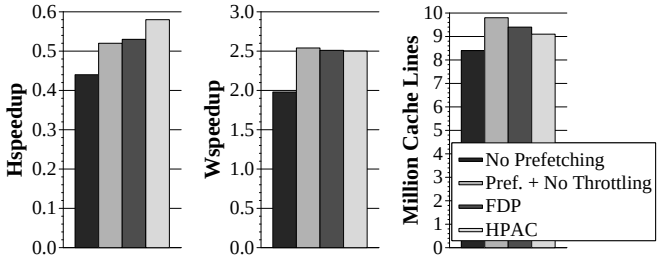
3.5.3 HPAC Performance with Different DRAM Scheduling Policies

We evaluate the performance of our proposal in a system with the recently proposed Prefetch-Aware DRAM Controller (PADC) [43]. PADC uses feedback about the accuracy of the prefetcher of each core to adaptively prioritize between prefetch requests of that prefetcher and demands in memory scheduling decisions. If the prefetcher of a core is accurate, prefetch requests from that core are treated with the same priority as demand requests. Otherwise, prefetches from that core are deprioritized below demands and prefetches from cores with high prefetch accuracy. Note that this local-only technique does not take into account inter-core interference caused by prefetchers. If the memory scheduler increases the priority of highly accurate but interfering prefetches, inter-core interference will likely increase. As a result, PADC cannot control the negative performance impact of accurate yet highly-interfering prefetchers in the memory system, which can degrade system performance.

Figure 3.11 shows the effect of HPAC when employed in a system with a prefetch-aware DRAM controller. HPAC increases the performance of a 4-core system that uses PADC by 12% (HS) on average while reducing bus traffic by 7%. HPAC's ability to reduce the negative interference caused by accurate prefetchers can have positive effects on PADC's options for better memory scheduling when PADC and HPAC are employed together. A reduction in interference caused by

one core's very aggressive prefetcher can reduce the number of demand misses of other cores. This removes many pollution-induced misses caused by the interfering core(s) and the new miss stream observed by the prefetchers of other cores can increase their accuracy significantly. HPAC's interference reduction enables PADC's memory scheduling decisions to take advantage of these more accurate prefetches. In contrast, PADC without HPAC would have seen inaccurate prefetch requests from such cores and deprioritized them due to their low accuracy. We conclude that systems with PADC-like memory controllers can benefit significantly if their prefetchers are controlled in a coordinated manner using HPAC.

The performance and bus traffic benefits of using HPAC with an FR-FCFS [65] memory scheduling policy are similar to those presented for the PAR-BS [55] fair memory scheduler which we use as our baseline (i.e., 12.4%/6.2% HS/WS improvement over FDP). We conclude that our proposal is orthogonal to the employed memory scheduling policy.



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 3.11: Performance of HPAC on system using PADC

3.5.4 Effect of HPAC on Fairness

Although HPAC's objective is to "improve system performance" not to "improve fairness," it is worth noting that HPAC's performance improvement does not come at the expense of fair treatment of all applications. We have evaluated HPAC's impact on performance unfairness [54] as defined in Section 3.4. Figure 3.12 shows that HPAC actually reduces unfairness in the system compared to all other techniques in both 4-core and 8-core systems. We found that this is because HPAC

significantly reduces the interference caused by applications that generate a very large number of prefetches on other less memory-intensive applications. This interference unfairly slows down the latter type of applications in the baseline since there is no mechanism that controls such interference.

We note that HPAC is orthogonal to techniques that provide fairness in shared resources [58, 32, 55]. As such, HPAC can be combined with techniques that are designed to provide fairness in shared multi-core resources. Note that we use Parallelism-Aware Batch Scheduling [55] as a fair memory scheduler in the *baseline* for all our evaluations. Figure 3.13 shows system performance and bus traffic of a 4-core system that uses a fair cache [58], a fair memory scheduler [55] and a state-of-the-art local-only prefetcher throttling mechanism (FDP) compared to 1) the combination of HPAC and a fair cache, and 2) HPAC by itself. Two observations are in order: First, using HPAC improves the performance of a system employing a fair cache. However, the improvement in performance is less than that obtained by HPAC alone. The reason is that constraining each core to a certain number of ways in each cache set as done in [58] reduces HPAC's flexibility. HPAC can throttle down a prefetcher that is causing large inter-core pollution to reduce such interference without the constraints of a fair cache [58]. Therefore HPAC can make more efficient use of cache space and perform better alone. Second, HPAC outperforms the combination of a fair cache, a fair memory scheduler, and FDP, by 10.2% (HS) and 4.7% (WS) while consuming 15% less bus traffic. We conclude that 1) our contribution is orthogonal to techniques that provide fairness in shared resources, and 2) the benefits of adjusting the aggressiveness of multiple prefetchers in a coordinated fashion (as done by HPAC) cannot be obtained by combining FDP, a fair cache, and a fair memory controller.

3.5.5 HPAC on Systems with Hardware Prefetch Filtering

Zhuang and Lee [75] propose a hardware-based prefetch filtering scheme that eliminates a prefetch request for an address if a prefetch request for the same address was useless in the past. They use a two-level branch predictor-like struc-

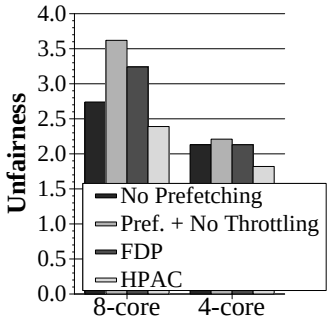


Figure 3.12: Unfairness in 8- and 4-core systems

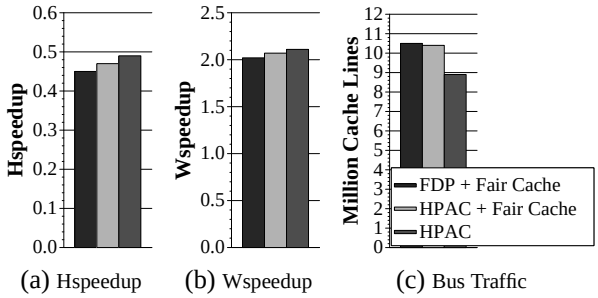
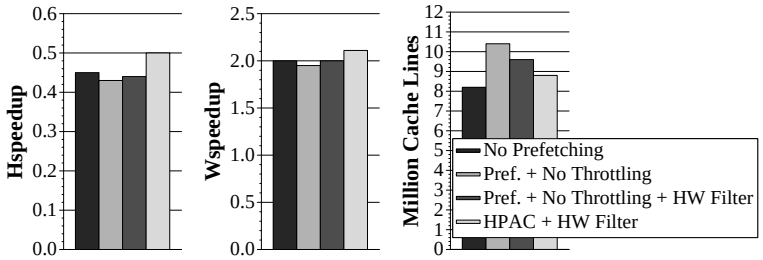


Figure 3.13: Comparison to combination of fair cache + fair memory scheduling + FDP

ture to record the usefulness of prefetches. We implemented HPAC on top of this hardware filtering scheme, and found that HPAC increases system performance by 12% while reducing bus traffic by 8.7% compared to hardware filtering alone on the evaluated 4-core workloads. Figure 3.14 shows that even though employing hardware prefetching on a 4-core system using aggressive prefetching does improve its performance and reduce bus traffic, system performance remains worse than that of a system with no prefetching on average. We conclude that even when hardware prefetch filtering is used, using HPAC makes prefetching much more effective on multi-core systems.

3.5.6 Multiple Types of Prefetchers per Core

Recent research suggests that by using "coordinated throttling" of multiple prefetchers of different types, hybrid prefetching systems can be useful [18]. Some current processors already employ more than one type of prefetcher on each core



(a) Hspeedup (b) Wspeedup (c) Bus Traffic

Figure 3.14: HPAC performance on 4-core system using HW prefetch filtering (all 32 workloads)

of a CMP [72]. We evaluate the effectiveness of our proposal on a 4-core system with two types of prefetcher per core and also with two different state-of-the-art local control policies as the local control for HPAC: FDP [67] and coordinated throttling [18]. Tables 3.9 and 3.10 show that HPAC is effective: 1) when multiple prefetchers of different types are employed within each core and 2) regardless of the local throttling policy used for prefetchers of each core. In all comparisons HPAC is the best performing of all schemes and produces the least bus traffic compared to any configuration with prefetching turned on.

	HS	WS	Bus Traffic
Δ over No Prefetching	7.9 %	5.1 %	10.7 %
Δ over Prefetching w. no Throttling	15.6 %	6.7 %	-13.9 %
Δ over FDP	10.6 %	3.2 %	-3 %

Table 3.9: Stream and GHB with HPAC (local policy: FDP)

	HS	WS	Bus Traffic
Δ over No Prefetching	6.3 %	4.0 %	12.2 %
Δ over Prefetching w. no Throttling	14.6 %	6.3 %	-12.7 %
Δ over coordinated throttling	12.2 %	4.5 %	-6.3 %

Table 3.10: Stream and GHB with HPAC
(local policy: coordinated throttling)

3.5.7 Sensitivity to System Parameters

We evaluate the sensitivity of our technique to three major memory system parameters: L2 cache size, memory latency and number of memory banks. Ta-

ble 3.11 shows the change in system performance (HS) and bus traffic provided by HPAC over FDP for each configuration. For these experiments we did not tune HPAC's parameters; doing so will likely increase HPAC's benefits even more. We conclude that our technique is effective for a wide variety of system parameters.

L2 Cache Size					
1 MB		2 MB		4 MB	
Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic
19.5%	-4%	10.7%	-3.2%	9.6%	-2.5%
Memory Latency - Latency per command (${}^tRP$, ${}^tRCD$, CL)					
13ns		15ns		17ns	
Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic
15 %	-3%	10.7%	-3.2%	6%	-3.4%
Number of Memory Banks					
8 banks		16 banks		32 banks	
Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic	Δ HS	Δ Bus Traffic
10.7%	-3.2%	12%	-1.5%	9%	-1%

Table 3.11: Effect of our proposal on Hspeedup (HS) and bus traffic with different system parameters on a 4-core system

3.5.8 Hardware Cost

Table 3.12 shows HPAC's required storage. The additional storage is 15.14KB (for a 4-core system), most of which is already required to implement FDP. This storage corresponds to 0.739% of the 2MB L2 baseline cache. The new global control structures require only 1.55KB of storage (for a 4-core system) on top of FDP. HPAC does not require any structures or logic that are on the critical path of execution.

3.6 Conclusion

We have proposed a low-cost technique that controls the aggressiveness of multiple prefetchers of different cores in chip-multiprocessors with the goal of improving system performance and making prefetching effective. We show that adjusting prefetcher aggressiveness using state-of-the-art techniques without paying attention to prefetcher-caused inter-core interference in shared memory sys-

Global Control	Closed form for N cores (bits)	N=4(bits)
Counters for global feedback	7 counters/core$\times$N$\times$16 bits/counter	448
Interference Pol. Filter per core	1024 entries $\times$ N $\times$ (pol. bit+(log N) bit proc. id)/entry	12,288
Local Control - FDP		
Proc. id for each L2 tag store entry	16384 blocks/Megabyte $\times$ S_{cache} $\times$ (log N) bit/block	65,536
Pref. bit for each L2 tag store entry	16384 blocks/Megabyte $\times$ S_{cache} $\times$ 1 bit/block	32,768
Pol. Filter for intra-core prefetch interference	1024 entries$\times$ N $\times$ (pol. bit+(log N) bit proc. id)/entry	12,288
Counters for local feedback	(8 counters/core$\times$N + 3 counters) $\times$16 bits/counter	560
Pref. bit per MSHR entry	32 entries/core $\times$ N $\times$ 1 bit/entry	128
Total storage	Sum of the above	15.14 KB

Table 3.12: Hardware cost of HPAC - Including both local and global throttling structures on an N-core CMP with S_{cache} MB L2 cache

tems can significantly degrade system performance compared to no prefetching at all. The key idea of our solution is to take into account prefetcher-caused inter-core interference in determining the aggressiveness of each core's prefetcher. Our scheme reduces the interference due to prefetchers using a coordinated control mechanism, thereby significantly improving system performance and bandwidth-efficiency compared to the state-of-the-art prefetcher control techniques that do not take into account such interference. We conclude that our technique significantly improves the performance of prefetching and makes it effective in multi-core environments.

Chapter 4

Fairness via Source Throttling

4.1 Introduction

When different applications concurrently execute on a CMP system, their memory requests can interfere with and delay each other in the shared memory subsystem. Compared to a scenario where each application runs alone on the CMP, this inter-core interference causes the execution of simultaneously running applications to slow down. However, sharing memory system resources affects the execution of different applications very differently because the resource management algorithms employed in the shared resources are unfair [54]. As a result some applications are unfairly slowed down significantly more than others .

Figure 4.1 shows two examples of vastly differing effects of resource-sharing on simultaneously executing applications on a 2-core CMP system (Section 4.4 describes our experimental setup). When *bzip2* and *art* run simultaneously with equal priorities, the inter-core interference caused by the sharing of memory system resources slows down *bzip2* by *5.2X* compared to when it is run alone while *art* slows down by only *1.15X*. In order to achieve system level fairness or quality of service (QoS) objectives, the system software (operating system or virtual machine monitor) expects proportional progress of *equal-priority* applications when running simultaneously. Clearly, disparities in slowdown like those shown in Figure 4.1 due to sharing of the memory system resources between simultaneously running equal-priority applications is unacceptable since it would make priority-based thread scheduling policies ineffective [20].

To mitigate this problem, previous papers [31, 36, 57, 54, 58, 32, 55] on fair memory system design for multi-core systems mainly focused on partitioning

Figure 4.1: Disparity in slowdowns due to unfairness

a particular shared resource (cache space, cache bandwidth, or memory bandwidth) to provide fairness in the use of that shared resource. However, none of these prior works directly target a *fair* memory system design that provides fair sharing of *all resources together*. We define a memory system design as *fair* if the slowdowns of equal-priority applications running simultaneously on the cores sharing that memory system are the same (this definition has been used in several prior papers [66, 49, 7, 22, 54]). This chapter shows that, employing separate uncoordinated fairness techniques together does not necessarily result in a fair memory system design. This is because fairness mechanisms in different resources can contradict each other. Our goal in this chapter is to develop a low-cost architectural technique that allows system software fairness policies to be achieved in CMPs by enabling fair sharing of the *entire memory system*, without requiring multiple complicated, specialized, and possibly contradictory fairness techniques for different shared resources.

Basic Idea: To achieve this goal, we propose a fundamentally new mechanism that 1) gathers dynamic feedback information about the unfairness in the system and 2) uses this information to dynamically adapt the rate at which the different cores inject requests into the shared memory subsystem such that system-level fairness objectives are met. To calculate unfairness at run-time, a slowdown value is estimated for each application in hardware. Slowdown is defined as T_{shared}/T_{alone}, where T_{shared} is the number of cycles it takes to run simultaneously with other applications and T_{alone} is the number of cycles it would have taken the application

to run alone. Unfairness is calculated as the ratio of the largest slowdown to the smallest slowdown of the simultaneously running applications. If the unfairness in the system becomes larger than the *unfairness threshold* set by the system software, the core that interferes most with the core experiencing the largest slowdown is throttled down. This means that the rate at which the most interfering core injects memory requests into the system is reduced, in order to reduce the inter-core interference it generates. If the system software's *fairness goal* is met, all cores are allowed to throttle up to improve system throughput while system unfairness is continuously monitored. This configurable hardware substrate enables the system software to achieve different QoS/fairness policies: it can determine the balance between fairness and system throughput, dictate different fairness objectives, and enforce thread priorities in the entire memory system.

4.2 Background and Motivation

We first present a brief background on how we model the shared memory system of CMPs. We then motivate our approach to providing fairness in the entire shared memory system by showing how employing resource-based fairness techniques does not necessarily provide better overall fairness.

4.2.1 Shared CMP Memory Systems

In this thesis, we assume that the last-level (L2) cache and off-chip DRAM bandwidth are shared by multiple cores on a chip as in many commercial CMPs [70, 72, 29, 1]. Each core has its own L1 cache. Miss Status Holding/information Registers (MSHRs) [39] keep track of all requests to the shared L2 cache until they are serviced. When an L1 cache miss occurs, an access request to the L2 cache is created by allocating an MSHR entry. Once the request is serviced by the L2 cache or DRAM system as a result of a cache hit or miss respectively, the corresponding MSHR entry is freed and used for a new request. Figure 4.2 gives a high level view of such a shared memory system. The number of MSHR entries for

a core indicates the total number of outstanding requests allowed to the L2 cache and DRAM system. Therefore increasing/decreasing the number of MSHR entries for a core can increase/decrease the rate at which memory requests from the core are injected into the shared memory system.

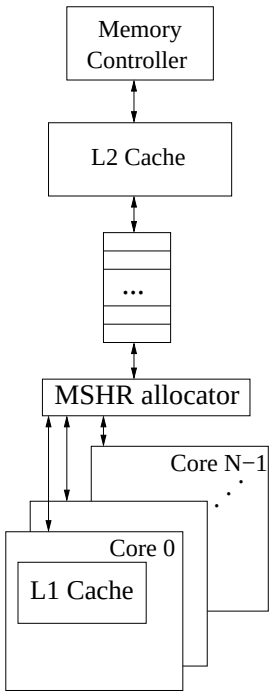


Figure 4.2: Shared CMP Memory System

4.2.2 Motivation

Most prior papers on providing fairness in shared resources focus on partitioning of a single shared resource. However, by partitioning *a single* shared resource, the demands on other shared resources may change such that neither system fairness nor system performance is improved. In the following example, we describe how constraining the rate at which an application's memory requests are injected to the shared resources can result in higher fairness and system performance than employing fair partitioning of a single resource.

Figure 4.3 shows the memory-related stall time[1] of equal-priority applications A and B either running alone on one core of a 2-core CMP (parts (a)-(d)), or, running concurrently with equal priority on different cores of a 2-core CMP (parts ((e)-(j)). For simplicity of explanation, we 1) assume that an application stalls when there is an outstanding memory request, 2) focus on requests going to the same cache set and memory bank, and 3) assume all shown accesses to the shared cache occur before any replacement happens. Application A is very memory-intensive, while application B is much less memory-intensive as can be seen by the different memory-related stall times they experience when running alone (Figures 4.3 (a)-(d)). As prior work has observed [55], when a memory-intensive application with already high memory-related stall time interferes with a less memory-intensive application with much smaller memory-related stall time, delaying the former improves system fairness because the additional delay causes a smaller slowdown for the memory-intensive application than for the non-intensive one. Doing so can also improve throughput by allowing the less memory-intensive application to quickly return to its compute-intensive portion while the memory-intensive application continues waiting on memory.

Figures 4.3(e) and (f) show the initial L2 cache state, access order and memory-related stall time when no fairness mechanism is employed in any of the shared resources. Application A's large number of memory requests arrive at the L2 cache earlier, and as a result, the small number of memory requests from application B are significantly delayed. This causes large unfairness because the compute-intensive application B is slowed down significantly more than the already-slow memory-intensive application A. Figures 4.3(g) and (h) show that employing a fair cache increases the fairness *in utilization of the cache* by allocating *an equal number of ways* from the accessed set to the two equal-priority applications. This increases

[1]Stall-time is the amount of execution time in which the application cannot retire instructions. Memory-related stall time caused by a memory request consists of: 1) time to access the L2 cache, and if the access is a miss 2) time to wait for the required DRAM bank to become available, and finally 3) time to access DRAM.

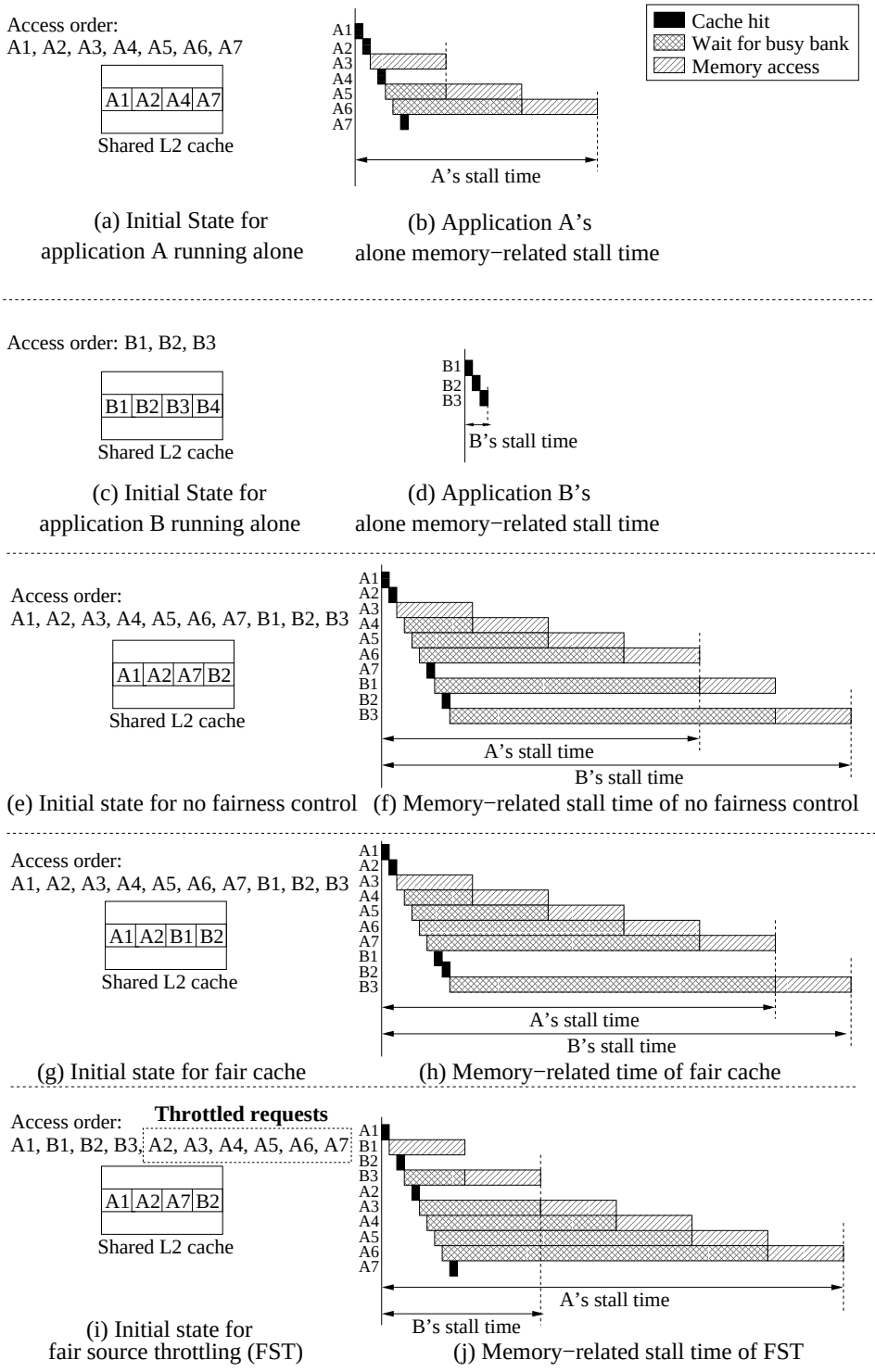


Figure 4.3: Access pattern and memory-related stall time of requests when application A running alone (a, b), application B running alone (c, d), A and B running concurrently with no fairness control (e, f), fair cache (g, h), and fair source throttling (i, j)

application A's cache misses compared to the baseline with no fairness control. Even though application B gets more hits as a result of fair sharing of the cache, its memory-related stall time does not reduce due to increased interference in the main memory system from application A's increased misses. Application B's memory requests are still delayed behind the large number of memory requests from application A. Application A's memory-related stall time increases slightly due to its increased cache misses, however, since application A already had a large memory-related stall time, this slight increase does not incur a large slowdown for it. As a result, fairness improves slightly, but system throughput degrades because the system spends more time stalling rather than computing compared to no fair caching.

In Figure 4.3, if the unfair slowdown of application B due to application A is detected at run-time, system fairness can be improved by limiting A's memory requests and reducing the frequency at which they are issued to the shared memory system. This is shown in the access order and memory-related stall times of Figures 4.3(i) and (j). If the frequency at which application A's memory requests are injected into the shared memory system is reduced, the access order changes as shown in Figure 4.3(i). We use the term *throttled requests* to refer to those requests from application A that are delayed when accessing the shared L2 cache due to A's reduced injection rate. As a result of the late arrival of these *throttled requests,* application B's memory-related stall time significantly reduces (because A's requests no longer interfere with B's) while application A's stall time increases slightly. Overall, this ultimately improves both system fairness and throughput compared to both no fairness control and just a fair cache. Fairness improves because the memory-intensive application is delayed without significantly increasing the less intensive application's memory related-stall time compared to when running alone. Delaying the memory-intensive application does not slow it down too much compared to when running alone, because even when running alone it has high memory-related stall time. System throughput improves because the total amount of time spent computing rather than stalling in the entire system increases, as can be seen by comparing the stall times in Figures 4.3 (f) and (h) to Figure 4.3 (j).

The **key insight** is that *both system fairness and throughput can improve by detecting high system unfairness at run-time and dynamically limiting the number of or delaying the issuing of memory requests from the aggressive applications*. In essence, we propose a new approach that performs *source-based* fairness in the entire memory system rather than *individual resource-based* fairness that implements complex and possibly contradictory fairness mechanisms in each resource. Sources (i.e., cores) can collectively achieve fairness by throttling themselves based on dynamic unfairness feedback. This eliminates the need for implementing possibly contradictory/conflicting fairness mechanisms and complicated coordination techniques between them.

4.3 Fairness via Source Throttling

To enable fairness in the entire memory system, we propose *Fairness via Source Throttling* (FST). The proposed mechanism consists of two major components: 1) *runtime unfairness evaluation* and 2) *dynamic request throttling*.

4.3.1 Runtime Unfairness Evaluation Overview

The goal of this component is to dynamically obtain an estimate of the unfairness in the CMP memory system. We use the definition of unfairness presented in Section 3.4.1.

The main challenge in the design of the runtime unfairness evaluation component is obtaining information about the number of cycles it would have taken an application to run alone, while it is running simultaneously with other applications. To do so, we estimate the number of *extra cycles* it takes an application to execute due to inter-core interference in the shared memory system, called T_{excess}. As defined in Section 3.4.1, T_{shared} is the number of cycles it takes to run simultaneously with other applications and T_{alone} is the number of cycles it would have taken the application to run alone on the same system. Given this, T_{alone} is estimated as $T_{shared} - T_{excess}$. Section 4.3.3 explains in detail how the runtime unfairness

evaluation component is implemented and in particular how T_{excess} is estimated. Assuming for now that this component is in place, we next explain how the information it provides is used to determine how each application is throttled to achieve fairness in the entire shared memory system.

4.3.2 Dynamic Request Throttling

This component is responsible for dynamically adjusting the rate at which each core/application[2] makes requests to the shared resources. This is done on an interval basis as shown in Figure 4.4.

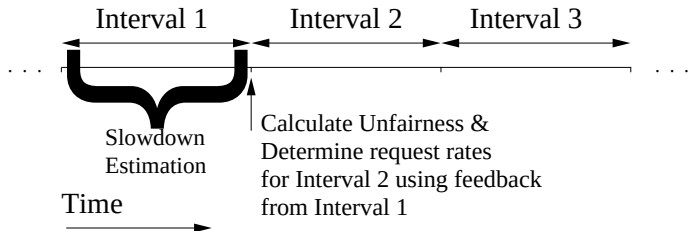


Figure 4.4: FST's interval-based estimation and throttling

An interval ends when each core has executed a certain number of instructions from the beginning of that interval. During each interval (for example *Interval 1* in Figure 4.4) the runtime unfairness evaluation component gathers feedback used to estimate the slowdown of each application. At the beginning of the next interval (*Interval 2*), the feedback information obtained during the prior interval is used to make a decision about the request rates of each application for that interval. More precisely, slowdown values estimated during *Interval 1* are used to estimate unfairness for the system. That unfairness value is used to determine the request rates for the different applications for the duration of *Interval 2*. During the next interval (*Interval 2*), those request rates are applied, and unfairness evaluation is performed again. The algorithm used to adjust the request rate of each application using the unfairness estimate calculated in the prior interval is shown in Algorithm 1. For clarity,

[2]Since each core runs a separate application, we use the words core and application interchangeably in this chapter.

Algorithm 1 is simplified for dual-core configurations. Section 4.3.5 presents the more general algorithm for more than two cores.

We define multiple possible levels of aggressiveness for the request rate of each application. The dynamic request throttling component makes a decision to increase/decrease or keep constant the request rate of each application at interval boundaries. We refer to increasing/decreasing the request rate of an application as throttling the application up/down.

Algorithm 1 Dynamic Request Throttling

```
if Estimated Unfairness > Unfairness Threshold then
    Throttle down application with the smallest slowdown (App_SmallestSlowdown)
    Throttle up application with the largest slowdown (App_LargestSlowdown)
    Reset Successive Fairness Achieved Intervals
else
    if Successive Fairness Achieved Intervals = threshold then
        Throttle all applications up
        Reset Successive Fairness Achieved Intervals
    else
        Increment Successive Fairness Achieved Intervals
    end if
end if
```

At the end of each interval, the algorithm compares the unfairness estimated in the previous interval to the unfairness threshold that is defined by system software. If the fairness goal has not been met in the previous interval, the algorithm reduces the request rate of the application with the smallest individual slowdown value (referred to as $App_{SmallestSlowdown}$) and increases the request rate of the application with the largest individual slowdown value (referred to as $App_{LargestSlowdown}$). This reduces the number and frequency of requests generated for and inserted into the memory resources by the application with the smallest estimated slowdown, thereby reducing its interference with other cores. The increase in the request rate of the application with the highest slowdown allows it to be more aggressive in exploiting Memory-Level Parallelism (MLP) [24] and as a result reduces its slowdown. If the fairness goal is met for a predetermined number of intervals (tracked by a $Successive\ Fairness\ Achieved\ Intervals$ counter in Al-

gorithm 1), the dynamic request throttling component attempts to increase system throughput by increasing the request rates of all applications by one level. This is done because our proposed mechanism strives to increase throughput while maintaining the fairness goals set by the system software. Increasing the request rate of all applications might result in unfairness. However, the unfairness evaluation during the interval in which this happens detects this occurrence and dynamically adjusts the requests rates again.

Throttling Mechanisms: Our mechanism increases/decreases the request rate of each application in multiple ways: 1) Adjusting the number of outstanding misses an application can have at any given time. To do so, an *MSHR quota*, which determines the maximum number of MSHR entries an application can use at any given time, is enforced for each application. Reducing MSHR entries for an application reduces the pressure caused by that application's requests on all shared memory system resources. This is done by limiting the number of concurrent requests from that application contending for service from the shared resources. This reduces other simultaneously running applications' memory-related stall times and gives them the opportunity to speed up. 2) Adjusting the *frequency at which requests in the MSHRs are issued to access L2*. Reducing this frequency for an application reduces the number of memory requests per unit time from that application which contend for shared resources. This mechanism is important for reducing the interference caused by applications that do not have high MLP to begin with. This is because such applications are not sensitive to a reduction in the number of MSHRs available to them. As such, throttling them just by reducing their MSHR quotas would not allow memory requests from other applications to be prioritized in accessing shared resources. We refer to this throttling technique as *frequency throttling*. We use both of these mechanisms to reduce the interference caused by $App_{SmallestSlowdown}$ on $App_{LargestSlowdown}$.

4.3.3 Unfairness Evaluation Component Design

T_{shared} is simply the number of cycles it takes to execute an application in an interval. Estimating T_{alone} is more difficult, and FST achieves this by estimating T_{excess} for each core, which is the number of cycles the core's execution time is lengthened due to interference from other cores in the shared memory system. To estimate T_{excess}, the unfairness evaluation component keeps track of inter-core interference each core incurs.

Tracking Inter-Core Interference: We consider three sources of inter-core interference: 1) cache, 2) DRAM bus and bank conflict, and 3) DRAM row-buffer.[3] Our mechanism uses an $InterferencePerCore$ bit-vector whose purpose is to indicate whether or not a core is delayed due to inter-core interference. In order to track interference from each source separately, a copy of $InterferencePerCore$ is maintained for each interference source. A main copy which is updated by taking the union of the different $InterferencePerCore$ vectors is eventually used to update T_{excess} as described below. When FST detects inter-core interference for core *i* at any shared resource, it sets bit *i* of the $InterferencePerCore$ bit-vector, indicating that the core was delayed due to interference. At the same time, it also sets an $InterferingCoreId$ field in the corresponding *interfered-with* memory request's MSHR entry. This field indicates which core interfered with this request and is later used to reset the corresponding bit in the $InterferencePerCore$ vector when the *interfered-with* request is scheduled/serviced. We explain this process in more detail for each resource below in Sections 4.3.3.1-4.3.3.3. If a memory request has not been interfered with, it's $InterferingCoreId$ will be the same as the core id of the core it was generated by.

Updating T_{excess}: FST stores the number of *extra cycles* it takes to execute a given interval's instructions due to inter-core interference (T_{excess}) in an

[3]On-chip interconnect can also experience inter-core interference [14]. Feedback information similar to that obtained for the three sources of inter-core interference we account for can be collected for the on-chip interconnect. That information can be incorporated into our technique seamlessly, which we leave as future work.

$ExcessCycles$ counter per core. Every cycle, if the $InterferencePerCore$ bit of a core is set, FST increments the corresponding core's $ExcessCycles$ counter.

Algorithm 2 shows how FST calculates $ExcessCycles$ for a given core *i*. The following subsections explain in detail how each source of inter-core interference is taken into account to set $InterferencePerCore$. Table 4.1 summarizes the required storage needed to implement the mechanisms explained here.

Algorithm 2 Estimation of T_{excess} for core *i*

Every cycle
  if $inter\text{-}core\ cache\ or\ DRAM\ bus\ or\ DRAM\ bank\ or\ DRAM\ row\text{-}buffer\ interference$ **then**
    set $InterferencePerCore$ bit *i*
    set $InterferingCoreId$ in delayed memory request
  end if
  if $InterferencePerCore$ bit *i* is set **then**
    Increment $ExcessCycles$ for core *i*
  end if

Every L2 cache fill for a miss due to interference OR
Every time a memory request which is a row-buffer miss due to interference is serviced
  reset $InterferencePerCore$ bit of core *i*
  $InterferingCoreId$ of core *i* = *i* (no interference)

Every time a memory request is scheduled to DRAM
  if Core *i* has no requests waiting on any bank which is busy servicing another core *j* (*j* != *i*) **then**
    reset $InterferencePerCore$ bit of core *i*
  end if

4.3.3.1 Cache Interference

In order to estimate inter-core cache interference, for each core *i* we need to track the last-level cache misses that are caused for core *i* by any other core *j*. To do so, FST uses a pollution filter for each core to approximate such misses. The pollution filter is a bit-vector that is indexed with the lower order bits of the ac-

cessed cache line's address.[4] In the bit-vector, a set entry indicates that a cache line belonging to the corresponding core was evicted by another core's request. When a request from core *j* replaces one of core *i*'s cache lines, core *i*'s filter is accessed using the evicted line's address, and the corresponding bit is set. When a memory request from core *i* misses the cache, its filter is accessed with the missing address. If the corresponding bit is set, the filter predicts that this line was previously evicted due to inter-core interference and the bit in the filter is reset. When such a prediction is made, once the interfered-with request is scheduled to DRAM the $InterferencePerCore$ bit corresponding to core *i* is set to indicate that core *i* is experiencing extra execution cycles due to cache interference. Once the interfered-with memory request is finished receiving service from the memory system and the corresponding cache line is filled, core *i*'s filter is accessed and the bit is reset and so is core *i*'s $InterferencePerCore$ bit.

4.3.3.2 DRAM Bus and Bank Conflict Interference

Inter-core DRAM bank conflict interference occurs when core *i*'s memory request cannot access the bank it maps to, because a request from some other core *j* is being serviced by that memory bank. DRAM bus conflict interference occurs when a core cannot use the DRAM because another core is using the DRAM bus. These situations are easily detected at the memory controller, as described in [54]. When such interference is detected, the $InterferencePerCore$ bit corresponding to core *i* is set to indicate that core *i* is stalling due to a DRAM bus or bank conflict. This bit is reset when no request from core *i* is being prevented access to DRAM by the other cores' requests.

[4]We empirically determined the pollution filter for each core to have 2K-entries in our evaluations.

4.3.3.3 DRAM Row-Buffer Interference

This type of interference occurs when a potential row-buffer hit of core *i* when running alone is converted to a row-buffer miss/conflict due to a memory request of some core *j* when running together with others. This happens if a request from core *j* closes a DRAM row opened by a prior request from core *i* that is also accessed by a subsequent request from core *i*. To track such interference, a *Shadow Row-buffer Address Register (SRAR)* is maintained for each core for each bank. Whenever core *i*'s memory request accesses some row *R*, the SRAR of core *i* is updated to row *R*. Accesses to the same bank from some other core *j* do not affect the SRAR of core *i*. As such, at any point in time, core *i*'s SRAR will contain the last row accessed by the last memory request serviced from that core in that bank. When core *i*'s memory request suffers a row-buffer miss because another core *j*'s row is open in the row-buffer of the accessed bank, the SRAR of core *i* is consulted. If the SRAR indicates a row-buffer hit would have happened, then inter-core row-buffer interference is detected. As a result, the $InterferencePerCore$ bit corresponding to core *i* is set. Once the memory request is serviced, the corresponding $InterferencePerCore$ bit is reset.[5]

4.3.3.4 Slowdown Due to Throttling

When an application is throttled, it experiences some slowdown due to the throttling. This slowdown is different from the inter-core interference induced slowdown estimated by the mechanisms of Sections 4.3.3.1 to 4.3.3.3. Throttling-induced slowdown is a function of an application's sensitivity to 1) the number of MSHRs that are available to it, 2) the frequency of injecting requests into the shared resources. Using profiling, we determine for each throttling level l, the corresponding slowdown (due to throttling) f of an application *A*. At runtime, any estimated slowdown for application *A* when running at throttling level l is multiplied by f. We

[5]To be more precise, the bit is reset "row buffer hit latency" cycles before the memory request is serviced. The memory request would have taken at least "row buffer hit latency" cycles had there been no interference.

find that accounting for this slowdown using this profiling information improves the system performance gained by FST by 4% on 4-core systems, as we show in Section 4.5.10.

Slowdown due to throttling can also be tracked by maintaining a counter for the number of cycles each application *A* stalls because it can not obtain an MSHR entry because of its limited *MSHR quota*. We separately keep track of the number of such cycles and refer to them as *excess cycles which are due to throttling* (as opposed to *excess cycles due to interference from other applications*). We discuss how this information is used later in a more general form of dynamic request throttling presented in Section 4.3.5, Algorithm 3.

4.3.3.5 Implementation Details

Section 4.3.3 describes how separate copies of $InterferencePerCore$ are maintained per interference source. The main copy which is used by FST for updating T_{excess} is physically located close by the L2 cache. Note that shared resources may be located far away from each other on the chip. Any possible timing constraints on the sending of updates to the $InterferencePerCore$ bit-vector from the shared resources can be eliminated by making these updates periodically, as we evaluate in Section 4.5.5.

4.3.4 System Software Support

Different Fairness Objectives: System-level fairness objectives and policies are generally decided by the system software (the operating system or virtual machine monitor). FST is intended as architectural support for enforcing such policies in shared memory system resources. The *fairness goal* to be achieved by FST can be configured by system software. To achieve this, we enable system software to determine the nature of the condition that triggers Algorithm 1. In the explanations of Section 4.3.2, the *triggering condition* is

$$Condition\ (1)\ \text{“}Estimated\ Unfairness > Unfairness\ Threshold\text{”}$$

System software might want to enforce different triggering conditions depending on the system's fairness/QoS requirements. To enable this capability, FST implements different triggering conditions from which the system software can choose. For example, the fairness goal that system software wants to achieve could be to keep the maximum slowdown of any given application below a threshold value. To enforce such a goal, the system software can configure FST such that the triggering condition in Algorithm 1 is changed to

$$Condition\ (2)\ \text{“}Estimated\ Slowdown_i > Max.\ Slowdown\ Threshold\text{”}$$

Thread Weights: So far, we have assumed all threads are of equal importance. FST can be seamlessly adjusted to distinguish between and provide differentiated services to threads with different priorities. We add the notion of *thread weights* to FST, which are communicated to it by the system software using special instructions. Higher slowdown values are more tolerable for less important or *lower weight* threads. To incorporate thread weights, FST uses *weighted slowdown* values calculated as:

$$WeightedSlowdown_i = Measured\ Slowdown_i \times Weight_i$$

By scaling the real slowdown of a thread with its weight, a thread with a higher weight appears as if it slowed down more than it really did, causing it to be favored by FST. Section 4.5.4 quantitatively evaluates FST with the above fairness goal and threads with different weights.

Thread Migration and Context Switches: FST can be seamlessly extended to work in the presence of thread migration and context switches. When a context switch happens or a thread is migrated, the interference state related to that thread is cleared. When a thread restarts executing after a context switch or migration, it starts at maximum throttle. The interference caused by the thread and the interference it suffers are dynamically re-estimated and FST adapts to the new set of co-executing applications.

4.3.5 General Dynamic Request Throttling

Scalability to More Cores: When the number of cores is greater than two, a more general form of Algorithm 1 is used. The design of the *unfairness evaluation* component for the more general form of Algorithm 1 is slightly different. This component gathers the following extra information for the more general form of dynamic request throttling presented in Algorithm 3: a) for each core *i*, FST maintains a set of *N-1* counters, where *N* is the number of simultaneously running applications. We refer to these *N-1* counters that FST uses to keep track of the amount of the inter-core interference caused by any other core *j* in the system for core *i* as $ExcessCycles_{ij}$. This information is used to identify which of the other applications in the system generates the most interference for core *i*, b) FST maintains the total inter-core interference an application on core *i* experiences due to interference from other cores in a $TotalExcessCyclesInterference_i$ counter per core, and c) as described in Section 4.3.3.4, those excess cycles that are caused as a result of an application being throttled down are accounted for separately in a $TotalExcessCyclesThrottling_i$ counter per core.

Algorithm 3 shows the generalized form of Algorithm 1 that uses the extra information described above to make more accurate throttling decisions in a system with more than two cores. The five most important changes are as follows:

First, when the algorithm is triggered due to unfair slowdown of core *i*, FST compares the $ExcessCycles_{ij}$ counter values for all cores $j \neq i$ to determine which other core is interfering most with core *i*. The core found to be the most interfering is throttled down. We do this in order to reduce the slowdown of the core with the largest slowdown value, and improve system fairness.

Second, first ready-first come first serve (FR-FCFS) [65] is a commonly used memory scheduling policy which we use in our baseline system. This memory scheduling policy has the potential to starve an application with no row-buffer locality in the presence of an application with high row-buffer locality (as discussed in prior work [57, 51, 54, 55]). Even when the interfering application is throttled

down, the potential for continued DRAM bank interference exists when FR-FCFS memory scheduling is used, due to the greedy row-hit-first nature of the schedul-

Algorithm 3 Dynamic Request Throttling - General Form

if $Estimated\ Unfairness > Unfairness\ Threshold$ AND
$App_{slow}\ slowdown / App_{interfering}\ slowdown > Unfairness\ Threshold$ **then**
  if App_{slow}'s excess cycles due to interference from $App_{interfering} > App_{slow}$'s $TotalExcessCyclesThrottling_i$ **then**
    Throttle down application that causes most interference ($App_{interfering}$) for application with largest $slowdown$
  end if
  Throttle up application with the largest $slowdown$ (App_{slow})
  Reset $Successive\ Fairness\ Achieved\ Intervals$
  Reset $Intervals\ To\ Wait\ To\ Throttle\ Up$ for $App_{interfering}$.

  // *Preventing bank service denial*
  if $App_{interfering}$ throttled lower than $Switch_{thr}$ AND causes greater than $Interference_{thr}$ amount of App_{slow}'s total interference **then**
    Temporarily stop prioritizing $App_{interfering}$ due to row hits in memory controller
  end if
  if $App_{RowHitNotPrioritized}$ has not been $App_{interfering}$ for $SwitchBack_{thr}$ intervals **then**
    Allow it to be prioritized in memory controller based on row-buffer hit status of its requests
  end if

  for all applications except $App_{interfering}$ and App_{slow} **do**
    if $Intervals\ To\ Wait\ To\ Throttle\ Up$ = $threshold1$ **then**
      throttle up
      Reset $Intervals\ To\ Wait\ To\ Throttle\ Up$ for this app.
    else
      Increment $Intervals\ To\ Wait\ To\ Throttle\ Up$ for this app.
    end if
  end for

else
  if $Successive\ Fairness\ Achieved\ Intervals = threshold2$ **then**
    Throttle up application with the smallest $slowdown$
    Reset $Successive\ Fairness\ Achieved\ Intervals$
  else
    Increment $Successive\ Fairness\ Achieved\ Intervals$
  end if
end if

ing algorithm: a throttled-down application with high row-buffer locality can deny service to another application continuously. To overcome this, we supplement FST with a heuristic that prevents this denial of service. Once an application has already been throttled down lower than $Switch_{thr}$%, if FST detects that this throttled application is generating greater than $Interference_{thr}\%$ of App_{slow}'s total interference, it will temporarily stop prioritizing the interfering application based on row-buffer hit status in the memory controller. We refer to this application as $App_{RowHitNotPrioritized}$. If $App_{RowHitNotPrioritized}$ has not been the most interfering application for $SwitchBack_{thr}$ number of intervals, its prioritization over other applications based on row-buffer hit status will be re-allowed in the memory controller. This is done because if an application with high row-buffer locality is not allowed to take advantage of row buffer hits for a long time, its performance will suffer.

Third, we change the condition based on which throttling triggers. Throttling triggers if both the following conditions hold: 1) the estimated unfairness ($Max.\ Slowdown/Min.\ Slowdown$) is greater than $Unfairness\ Threshold$ and, 2) the ratio between the slowdowns of the core with the largest slowdown (App_{slow}) and the core generating the most interference ($App_{interfering}$) is greater than $Unfairness\ Threshold$. Doing so helps reduce excessive throttling when two applications significantly interfere with eachother and alternate between being identified as App_{slow} and $App_{interfering}$. By comparing their slowdowns before throttling is performed, overall throughput is improved by avoiding excessive throttling.

Fourth, we restrict throttling down of $App_{interfering}$ to cases where the slowdown that App_{slow} is suffering is mainly caused by inter-core interference and is not a result of App_{slow} having been throttled down in previous intervals. We do this because we observe that there are situations where an application suffers slowdown that is incurred as a result of throttling from previous intervals. If the excess cycles that App_{slow} suffers due to not being able to acquire MSHR entries is greater than the excess cycles caused for it by $App_{interfering}$ we do not throttle down $App_{interfering}$ as this would result in a loss of throughput. In such cases the detected

unfairness can be resolved by throttling up App_{slow} and reducing its slowdown by allowing it to acquire more MSHR entries.

Fifth, cores that are neither the core with the largest slowdown (App_{slow}) nor the core generating the most interference ($App_{interfering}$) for the core with the largest slowdown are throttled up every *threshold1* intervals. This is a performance optimization that allows cores to be aggressive if they are not the main contributors to the unfairness in the system.

4.3.6 Hardware Cost and Implementation Details

Table 4.1 shows the breakdown of FST's required storage. The total storage cost required by our implementation of FST is 11.24KB which is only 0.55% the size of the L2 cache being used. FST does not require any structure or logic that is on the critical path since all updates to interference-tracking structures can be made periodically at relatively large intervals to eliminate any timing constraints (see Section 4.5.5).

Figure 4.5 shows the shared CMP memory system we model for evaluation of FST including additional structures for tracking interference added to the baseline memory system shown in Figure 4.2. The two boxes on the right of the figure contain interference tracking structures and counters, and the shaded bit positions in the L2 cache lines and MSHR entries on the left are additions to these structures required by FST.

4.3.7 Lightweight FST

In this section we describe an alternative FST implementation that requires less hardware and is more scalable. In this alternative implementation, we do not keep track of how much interference is caused by each application for each other application which requires N^2 *ExcessCycles* counters, as described in the previous subsection. Instead, we propose maintaining two counters for each core *i*. One counter tracks the total number of *ExcessCycles* that the application executing on

	Cost for N cores	Cost for N = 4
$ExcessCycles$ counters	N × N × 16 bits/counter	256 bits
Interference pollution filter per core	2048 entries × N × (1 pollution bit + (log_2 N) bit processor id)/entry	24,576 bits
$InterferingCoreId$ per MSHR entry	32 entries/core× N × 2 interference sources × (log_2 N) bits/entry	512 bits
$InterferencePerCore$ bit-vector	3 interference sources × N × N × 1 bit	48 bits
Shadow row-buffer address register	N × # of DRAM banks (B) × 32 bits/address	1024 bits
$Successive\ Fairness\ Achieved\ Intervals$ counter $Intervals\ To\ Wait\ To\ Throttle\ Up$ counter per core $Inst\ Count\ Each\ Interval$ per core	(2 × N + 1) × 16 bits/counter	144 bits
Core id per tag store entry in K MB L2 cache	16384 blocks/Megabyte × K × (log_2 N) bit/block	65,536 bits
Total hardware cost for N-core system	Sum of the above	92092(**11.24 KB**)
Percentage area overhead (as fraction of the baseline K MB L2 cache)	Sum (KB) × 100 / (K × 1024)	11.24KB/2MB = **0.55%**

Table 4.1: Hardware cost of FST on a 4-core CMP system

core *i* generated for *any other* concurrently-executing application. We refer to this counter as $ExcessCyclesGenerated_i$. The other counter tracks the total number of *ExcessCycles* that *any other* concurrently-executing application creates for the application on core *i*. We refer to this counter as $ExcessCyclesSuffered_i$. This requires a total of $2N$ 16-bit counters to be maintained and makes for a more scalable solution with larger numbers of cores.

For the lightweight FST implementation to work with the counters described above we modify Algorithm 3 as follows. With lightweight FST, the core executing the application that has the largest slowdown App_{slow} is still throttled up. However, as opposed to throttling down the core executing the application which causes the most interference for App_{slow} (i.e., $App_{interfering}$) in Algorithm 3, we throttle down the core that is executing the application which is generating the most interference for other concurrently-executing applications. This is the core with the highest $ExcessCyclesGenerated_i$ counter in a given interval. We evaluate the performance of our lightweight FST in Section 4.5.7.

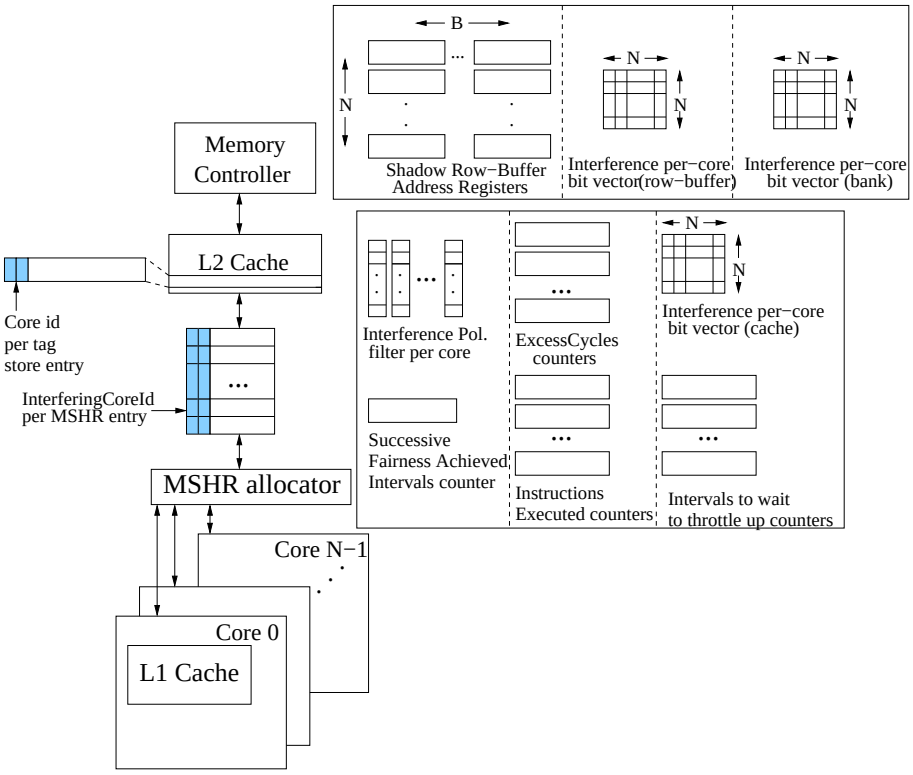


Figure 4.5: Changes made to the memory system

4.4 Methodology

4.4.1 Metrics

To measure CMP system performance, we use *Harmonic mean of Speedups (Hspeedup)* [49], *Weighted Speedup (Wspeedup)* [66], and *Individual Speedup (IS)*, which are defined in Section 3.4.1. Since *Hspeedup* provides a balanced measure between fairness and system throughput as shown in previous work [49], we use it as our primary evaluation metric. In order to demonstrate fairness improvements, we report *Unfairness* (see Section 3.4.1), as defined in [22, 54]. We also report *Maximum Slowdown* to evaluate fairness improvements, which is the maximum individual slowdown that any application in a workload experiences. *Maximum Slowdown* is an indicator of the minimum service that any application in the workload receives.

4.4.2 Processor Model

Table 4.2 shows the baseline configuration of each core and the shared resource configuration for the 2 and 4-core CMP systems we use in the evaluations of this chapter. We faithfully model all port contention, queuing effects, bank conflicts, and other major DDR3 DRAM system constraints in the memory subsystem.

Execution Core	15 stage out of order processor Decode/retire up to 4 instructions Issue/execute up to 8 micro instructions 256-entry reorder buffer
Front End	Fetch up to 2 branches; 4K-entry BTB 64K-entry Hybrid branch predictor
On-chip Caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line L1 D-cache: 32KB, 4-way, 2-cycle, 64B line Shared unified L2: 1MB (2MB for 4-core), 8-way (16-way for 4-core), 16-bank, 15-cycle (20-cycle for 4-core), 1 port, 64B line size
DRAM Controller	On-chip, FR-FCFS scheduling policy [65] 128-entry MSHR and memory request buffer
DRAM and Bus	667MHz bus cycle, DDR3 1333MHz [50] 8B-wide data bus, 8 DRAM banks, 16KB row buffer per bank Latency: 15-15-15ns (^{t}RP-^{t}RCD-CL), corresponds to 100-100-100 processor cycles Round-trip L2 miss latency: Row-buffer hit: 36ns, conflict: 66ns

Table 4.2: Baseline system configuration

4.4.3 Workloads

We use the SPEC CPU 2000/2006 benchmarks for our evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. We ran each benchmark with the reference input set for 200 million x86 instructions selected by Pinpoints [62] as a representative portion for the 2-core experiments. Due to long simulation times, 4-core experiments were done with 50 million instructions per benchmark.

We classify benchmarks as *highly memory-intensive/with medium memory intensity/non-intensive* for our analyses and workload selection. We refer to a benchmark as highly memory-intensive if its L2 Cache Misses per 1K Instructions

(MPKI) is greater than ten. If the MPKI value is greater than one but less than ten, we say the benchmark has medium memory-intensity. If the MPKI value is less than one, we refer to it as non-intensive. This classification is based on measurements made when each benchmark was run alone on the 2-core system. Table 4.3 shows the characteristics of the benchmarks that appear in the evaluated workloads when run on the 2-core system.

Benchmark	Type	IPC	MPKI	Benchmark	Type	IPC	MPKI
art	FP00	0.10	90.89	milc	FP06	0.30	29.33
soplex	FP06	0.28	21.24	leslie3d	FP06	0.41	20.88
lbm	FP06	0.45	20.16	bwaves	FP06	0.46	18.71
GemsFDTD	FP06	0.46	15.63	lucas	FP00	0.61	10.61
astar	INT06	0.37	10.19	omnetpp	INT06	0.36	10.11
mgrid	FP00	0.52	6.5	gcc	INT06	0.45	6.26
zeusmp	FP06	0.82	4.69	cactusADM	FP06	0.60	4.51
bzip2	INT06	1.14	2.61	xalancbmk	INT06	0.71	1.68
h264ref	INT06	1.46	1.28	vortex	INT00	1.01	1.24
parser	INT00	1.24	0.91	apsi	FP00	1.81	0.85
ammp	FP00	1.8	0.75	perlbench	INT06	1.49	0.68
mesa	FP00	1.82	0.61	gromacs	FP06	1.06	0.29
namd	FP06	2.25	0.18	crafty	INT00	1.82	0.1
calculix	FP06	2.28	0.05	gamess	FP06	2.32	0.04
povray	FP06	1.88	0.02	-	-	-	-

Table 4.3: Characteristics of 29 SPEC 2000/2006 benchmarks: IPC and MPKI (L2 cache Misses Per 1K Instructions)

We used 18 two-application and 10 four-application multi-programmed workloads for our 2-core and 4-core evaluations respectively. The 2-core workloads were chosen such that at least one of the benchmarks is highly memory intensive. For this purpose we used either *art* from SPEC2000 or *lbm* from SPEC2006. For the second benchmark of each 2-core workload, applications of different memory intensity were used in order to cover a wide range of different combinations. Of the 18 benchmarks combined with either *art* or *lbm*, seven benchmarks have high memory intensity, six have medium intensity, and five have low memory intensity. The ten 4-core workloads were randomly selected with the condition that the evaluated workloads each include at least one benchmark with high memory intensity and at least one benchmark with medium or high memory intensity.

4.4.4 FST Parameters Used in Evaluation

Table 4.4 shows the FST parameter values we use in our evaluation unless stated otherwise. There are eight aggressiveness levels used for the request rate of each application: 2%, 3%, 4%, 5%, 10%, 25%, 50% and 100%. These levels denote the scaling of the MSHR quota and the request rate in terms of percentage. For example, when FST throttles an application to 5% of its total request rate on a system with 128 MSHRs, two parameters are adjusted. First, the application is given a 5% quota of the total number of available MSHRs (in this case, 6 MSHRs). Second, the application's memory requests in the MSHRs are issued to access the L2 cache at 5% of the maximum possible frequency (i.e., once every 20 cycles).

$Unfairness\ Threshold$	$Successive\ Fairness\ Achieved\ Intervals\ Threshold$	$Intervals\ Wait\ To\ Throttle\ Up$	$Interval\ Length$
1.4	4	2	25Kinsts

$Switch_{thr}$	$Interference_{thr}$	$SwitchBack_{thr}$
5%	70%	3 intervals

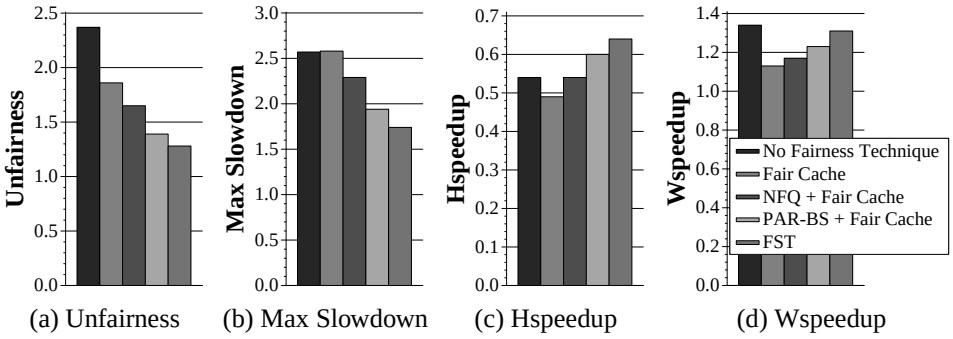
Table 4.4: FST parameters

4.5 Experimental Evaluation

We evaluate our proposed techniques on both 2-core (Section 4.5.1) and 4-core systems (all other sections). We compare FST to four other systems in our evaluations: 1) a baseline system with no fairness techniques employed in the shared memory system, using LRU cache replacement and FR-FCFS memory scheduling [65], both of which have been shown to be unfair [36, 57, 51]. We refer to this baseline as *NoFairness*, 2) a system with only fair cache capacity management using the virtual private caches technique [58], called *FairCache*, 3) a system with a network fair queuing (NFQ) fair memory scheduler [57] combined with fair cache capacity management [58], called *NFQ+FairCache*, 4) a system with a parallelism-aware batch scheduling (PAR-BS) fair memory scheduler [55] combined with fair cache capacity management [58], called *PAR-BS+FairCache*.

4.5.1 2-core System Results

Figure 4.6 shows system performance and unfairness averaged (using geometric mean) across 18 workloads evaluated on the 2-core system. Figure 4.7 shows the Hspeedup performance of FST and other fairness techniques normalized to that of a system without any fairness technique for each of the 18 evaluated 2-core workloads. FST provides the highest system performance (in terms of Hspeedup) and the best unfairness among all evaluated techniques. We make several key observations:



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

Figure 4.6: Average performance of FST on the 2-core system

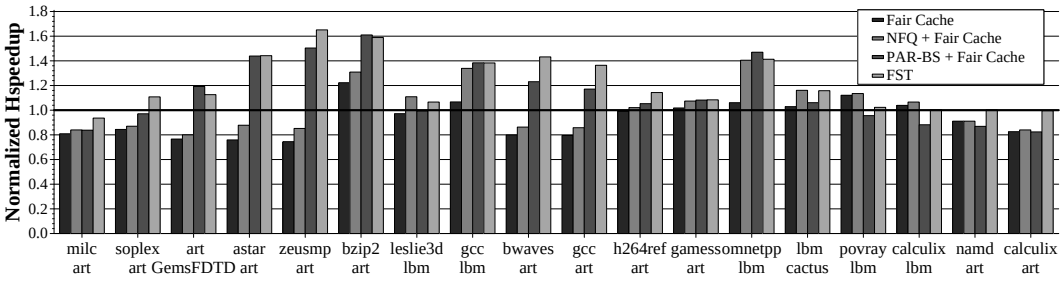


Figure 4.7: Hspeedup of 18 2-core workloads normalized to no fairness control

1. Fair caching's unfairness reduction comes at the cost of a large degradation in system performance. Also average maximum slowdown, which indicates the most any application in a workload is slowed down due to sharing of memory system resources, is increased slightly. These happen because fair caching changes the memory access patterns of applications. Since the memory access scheduler is unfair, the fairness benefits of the fair cache itself are reverted by the memory scheduler.

2. NFQ+FairCache together reduces system unfairness by 30.2% compared to *NoFairness* and reduces maximum slowdown by 10.9%. However, this degrades Wspeedup (by 12.3%). The combination of PAR-BS and fair caching improves both system performance and fairness compared to the combination of NFQ and a fair cache. The main reason is that PAR-BS preserves both DRAM bank parallelism and row-buffer locality of each thread better than NFQ, as shown in previous work [55]. Compared to the baseline with no fairness control, employing PAR-BS and a fair cache reduces unfairness and maximum slowdown by 41.3%/24.5% and improves Hspeedup by 11.5%. However, this improvement comes at the expense of a (7.8%) Wspeedup degradation.

NFQ+FairCache and PAR-BS+FairCache both significantly degrade system throughput (Wspeedup) compared to employing no fairness mechanisms. This is due to two reasons both of which lead to the delaying of memory non-intensive applications (Recall that prioritizing memory non-intensive applications is better for system throughput [57, 55]). First, the fairness mechanisms that are employed separately in each resource interact negatively with each other, leading to one mechanism (e.g. fair caching) increasing the pressure on the other (fair memory scheduling). As a result, even though fair caching might benefit system throughput by giving more resources to a memory non-intensive application, increased misses of the memory-intensive application due to fair caching causes more congestion in the memory system, leading to both the memory-intensive and non-intensive applications to be delayed. Second, even though the combination of a fair cache and a fair memory controller can prioritize a memory non-intensive application's requests, this prioritization can be temporary. The deprioritized memory-intensive application can still fill the shared MSHRs with its requests, thereby denying the non-intensive application entry into the memory system. Hence, the non-intensive application stalls because it cannot inject enough requests into the memory system. As a result, the memory non-intensive application's performance does not improve while the memory-intensive application's performance degrades (due to fair caching), resulting in system throughput degradation.

3. FST reduces system unfairness and maximum slowdown by 46.1%/32.3% while also improving Hspeedup by 20% and degrades Wspeedup by 1.8% compared to *NoFairness.* Unlike other fairness mechanisms, FST improves both system performance and fairness, without large degradation to Wspeedup. This is due to two major reasons. First, FST provides a coordinated approach in which both the cache and the memory controller receive less frequent requests from the applications causing unfairness. This reduces the starvation of the applications that are unfairly slowed down as well as interference of requests in the memory system, leading to better system performance for almost all applications. Second, because FST uses *MSHR quotas* to limit requests injected by memory-intensive applications that cause unfairness, these memory-intensive applications do not deny other applications' entry into the memory system. As such, unlike other fairness techniques that do not consider fairness in memory system buffers (e.g., MSHRs), FST ensures that unfairly slowed-down applications are prioritized in the entire memory system, including all the buffers, caches, and schedulers.

Table 4.5 summarizes our results for the 2-core evaluations. Compared to the previous technique that provides the highest system throughput (i.e. NoFairness), FST provides a significantly better balance between system fairness and performance. Compared to the previous technique that provides the best fairness (PAR-BS+FairCache), FST improves both system performance and fairness. We conclude that FST provides the best system fairness as well as the best balance between system fairness and performance.

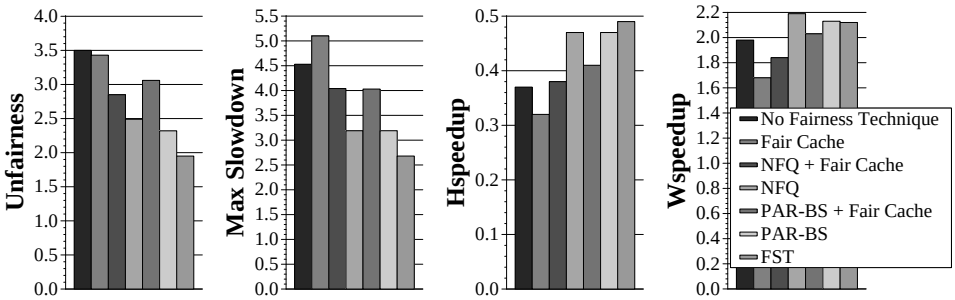
	Unfairness	Maximum Slowdown	Hspeedup	Wspeedup
FSTΔ over No Fairness Mechanism	-46.1%	-32.3%	20%	-1.8%
FSTΔ over Fair Cache	-31.3%	-32.6%	30.2%	16.1%
FSTΔ over NFQ + Fair Cache	-22.8%	-24.1%	19.7%	11.9%
FSTΔ over PAR-BS + Fair Cache	-8.2%	-10.4%	7.5%	6.4%

Table 4.5: Summary of results on the 2-core system

4.5.2 4-core System Results

4.5.2.1 Overall Performance

Figure 4.8 shows unfairness and system performance averaged across the ten evaluated 4-core workloads. FST provides the best fairness (in terms of both smallest unfairness and smallest maximum slowdown) and Hspeedup among all evaluated techniques,[6] while providing Wspeedup that is within 3.5% that of the best previous technique. Overall, FST reduces unfairness and maximum slowdown by 44.4%/41%[7] and increases system performance by 30.4% (Hspeedup) and 6.9% (Wspeedup) compared to *NoFairness*. Compared to NFQ, the previous technique with the highest system throughput (Wspeedup), FST reduces unfairness and max slowdown by 22%/16.1% and increases Hspeedup by 4.2%. FST's large performance improvement is mainly due to the large reduction in unfairness.[8]



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

Figure 4.8: Average performance of FST on the 4-core system

Note that the overall trends in the 4-core system are similar to those in the 2-core system except that previous fairness mechanisms do not significantly improve

[6] In this subsection we also include data points for NFQ alone and PAR-BS alone with no FairCache to show how the uncoordinated combination of fairness techniques at different shared resources can result in degradation of both performance and fairness compared to when only one is employed.

[7] Similarly, FST also reduces the coefficient of variation, an alternative unfairness metric, by 45%.

[8] Since relative slowdowns of different applications are most important to improving unfairness and performance using FST, highly accurate T_{excess} estimations are not necessary for such improvements. However, we find that with the mechanisms proposed in this chapter the application which causes the most interference for the most-slowed-down application is on average identified correctly in 70% of the intervals.

fairness in the 4-core system. As we will explain in detail in Section 4.5.2.2, this is due to prioritization of non-intensive applications in individual resources by previous fairness mechanisms regardless of whether or not such applications are actually slowed down.

Figure 4.9 shows the harmonic speedup performance of FST and other fairness techniques normalized to that of a system without any fairness technique for each of the ten workloads. Figure 4.10 shows the system unfairness of all the techniques for each of the ten workloads. We make two major conclusions. First, FST improves system performance (both Hspeedup and Wspeedup) and fairness compared to no fairness control for all workloads. Second, FST provides the best trade-off between system performance and system fairness: FST has the highest Hspeedup compared to the previous technique with the highest average system performance (NFQ) on seven of the ten workloads, and the best fairness compared to the previous technique with the best system fairness (PAR-BS) on seven of the ten workloads.

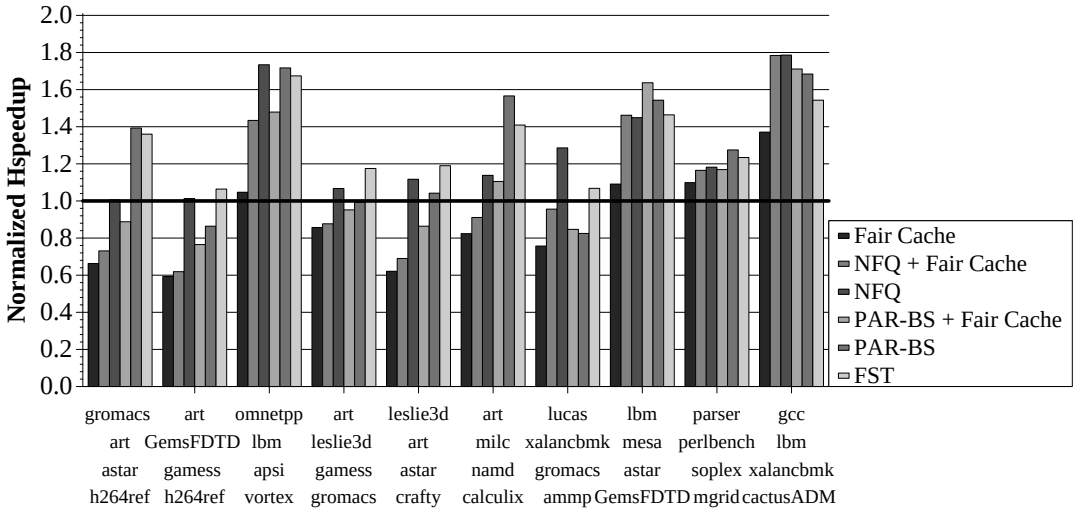


Figure 4.9: Normalized speedup of ten 4-core workloads

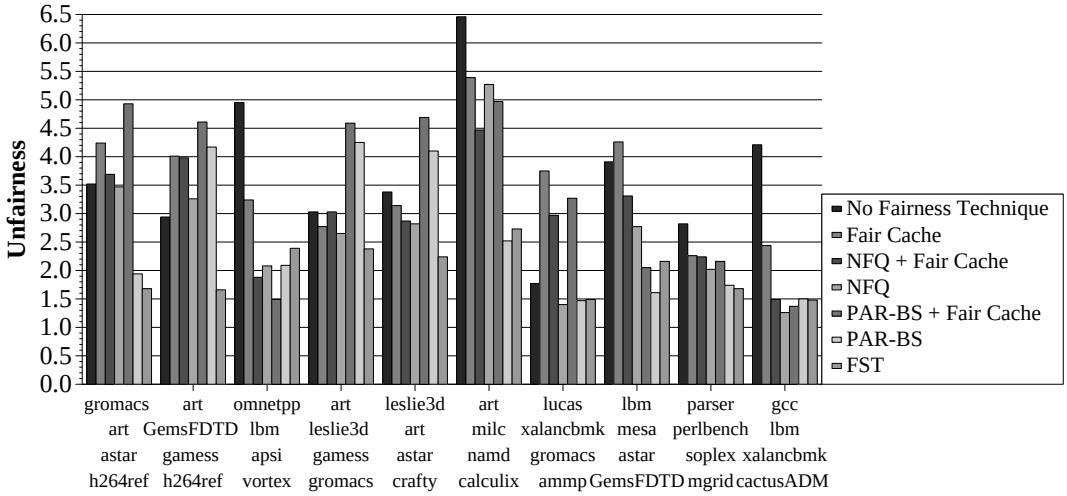


Figure 4.10: Unfairness of ten 4-core workloads

4.5.2.2 Case Study

To provide more insight into the performance and fairness improvements of FST, we analyze one 4-core workload in detail. This workload is a mix of applications of different levels of memory intensity. *Art* and *leslie* are both highly memory-intensive, while *gamess* and *gromacs* are non-intensive (as shown in Table 4.3). When these applications are run simultaneously on a 4-core system with no fairness control, the two memory-intensive applications (especially *art*) generate a large amount of memory traffic. *Art*'s large number of memory requests to the shared resources unfairly slows down the other three applications, while *art* does not slow down significantly. Figures 4.11 and 4.12 show individual benchmark performance and system performance/fairness, respectively (note that Figure 4.11 shows speedup over the alone run which is the inverse of individual slowdown, defined in Section 3.4.1). Several observations are in order:

1. NFQ+FairCache significantly degrades system performance by 12.3% (Hspeedup) and 7.1% (Wspeedup) compared to no fairness control. This combination slows down the memory-intensive applications too much, resulting in a 16.7% increase in maximum slowdown compared to employing no fairness technique. The largest slowdowns are experienced by the memory-intensive *art* and

leslie because they both get less cache space due to FairCache and are deprioritized in DRAM due to NFQ. On the other hand, when NFQ alone is employed, the memory non-intensive application's performance is slightly improved by prioritizing them in DRAM at small reductions to the performance of the memory-intensive applications. NFQ alone improves system performance by 6.7%/3.1% (HS/WS) and reduces unfairness/maximum slowdown by 12.7%/10.9%. However these gains are not large even though there is significant interference in the memory system for this workload because NFQ does not address interference caused in the shared cache.

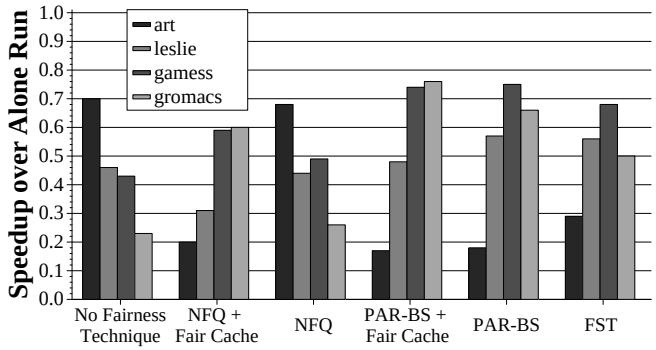
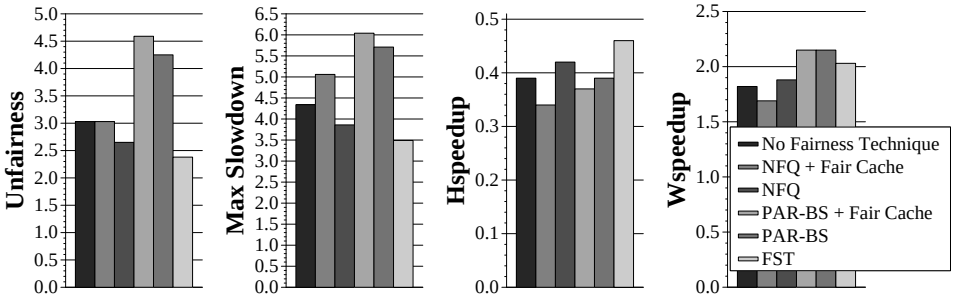


Figure 4.11: Case Study - individual application behavior



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

Figure 4.12: Case study - system behavior

2. With PAR-BS+FairCache and PAR-BS, *art* is heavily deprioritized while the performance of the less memory intensive applications is improved unfairly. This results in improved overall system throughput (WSpeedup). These two techniques are an example of where unfair treatment of applications in a workload

may increase system throughput at the cost of large increases to unfairness and maximum slowdown (51.5%/39% and 40.4%/31.6% for PAR-BS+FairCache and PAR-BS respectively). Average system turnaround time (Hspeedup) also degrades compared to not using any fairness technique. These techniques overly deprioritize memory intensive applications (specifically *art*) because they do not explicitly detect when such applications cause slowdowns for others. They prioritize non-intensive applications almost all the time, regardless of whether or not they are actually slowed down in the memory system. In contrast, our approach explicitly detects when memory-intensive applications are causing unfairness in the system. If they are not causing unfairness, FST does not deprioritize them. As a result, their performance is not unnecessarily reduced. This effect is observed by examining the most memory-intensive application's (*art*'s) performance with FST. With FST, *art* has higher performance than with any of the other fairness techniques.

3. FST increases system performance by 17.5%/11.6% (HS/WS) while reducing unfairness/maximum slowdown by 21.4%/19.5% compared to no fairness control. In this workload, the memory-intensive *art* and *leslie* cause significant interference to each other in all shared resources and to *gromacs* in the shared cache. Unlike other fairness techniques, FST dynamically tracks the interference and the unfairness in the system in a fine-grained manner. When the memory-intensive applications are causing interference and increasing unfairness, FST throttles the offending *hog* application(s). In contrast, when the applications are not interfering significantly with each other, FST allows them to freely share resources in order to maximize each application's performance. The fine-grained dynamic detection of unfairness and enforcement of fairness mechanisms only when they are needed allow FST to achieve higher system performance (Hspeedup) and a better balance between fairness and performance than other techniques.

To provide insight into the dynamic behavior of FST, Figure 4.13 shows the percentage of time each core spends at each throttling level. FST significantly throttles down *art* and *leslie* much of the time (but not always) to reduce the inter-core interference they generate for each other and the less memory intensive applica-

tions. As a result, *art* and *leslie* spend almost 25%/30% of their execution time at 10% or less of their full aggressiveness. Also, a lot of the time, *art* can prevent bank service to the accesses of *leslie* to the same bank. FST detects this and disallows *art*'s requests to be prioritized based on row-buffer hits for 74% of all intervals, preventing *art* from causing bank service denial as described in Section 4.3.5. Note that *art* spends approximately 55% of its time at throttling level 100, which shows that FST detects times when art is not causing large interference and does not penalize it. Figure 4.13 also shows that FST detects interference caused by not only *art* but also other applications. *leslie*, *gromacs*, and even *gamess* are detected to generate inter-core interference for other applications in certain execution intervals. As such, FST dynamically adapts its fairness control decisions to the interference patterns of applications rather than simply prioritizing memory non-intensive applications. Therefore, unlike other fairness techniques, FST does not overly deprioritize *art* in the memory system.

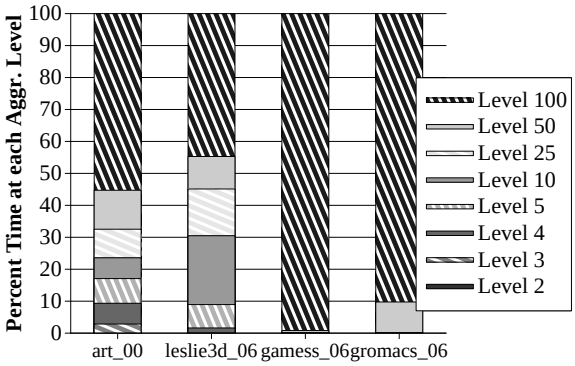


Figure 4.13: Case study - application throttling levels

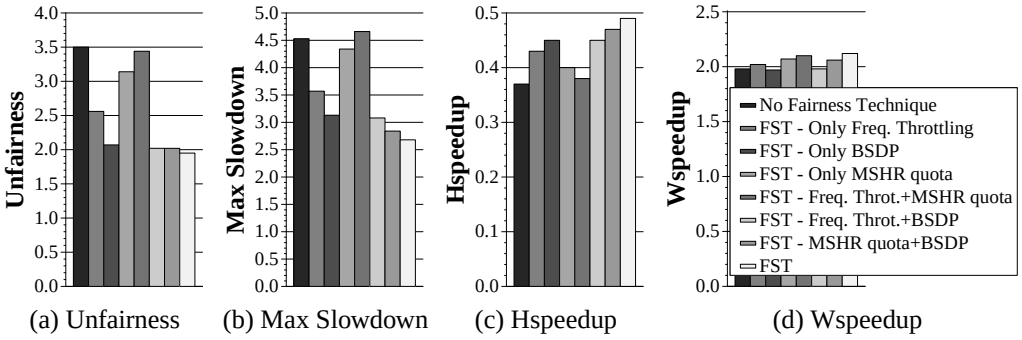
We conclude that FST provides a higher-performance approach to attaining fairness than coarsely tracking the memory-intensity of applications and deprioritizing memory-intensive applications without dynamic knowledge of interference and unfairness. FST achieves this by tracking unfairness in the system and making fairness/throttling decisions based on that tracking in a finer-grained manner.

4.5.3 Effect of Throttling Mechanisms

As described in Section 4.3.2, FST uses a combination of two mechanisms to throttle an application up/down and increase/decrease its request rate from the shared resources: 1) Applying an *MSHR quota* to each application, 2) Adjusting the frequency at which requests in the MSHRs are issued to access the L2. Section 4.3.5 explains how to prevent bank service denial from FR-FCFS memory scheduling within FST. Figure 4.14 shows the effect of each of the different throttling mechanisms, the effect of bank service denial prevention (BSDP), and FST on the 4-core system. Several observations are in order:

1. Employing BSDP always improves performance regardless of the throttling mechanism being used. BSDP's improvements are due to resolution of a problem we refer to as the *over-throttling problem*. As explained in Section 4.3.5, even throttled applications can cause significant interference when the memory controller uses an FR-FCFS scheduling algorithm. When this occurs (using the terminology of Section 4.3.5), FST detects some already throttled down application to be $App_{interfering}$ and continuously throttles it down further because the estimated unfairness remains high and App_{slow} stays the same. We call this *over-throttling* of $App_{interfering}$. BSDP resolves this issue by eliminating the cause of bank service denial due to FR-FCFS scheduling.

In Figure 4.14, the fourth and fifth bars from the left in each subgraph show the importance of BSDP. Without BSDP, enabling MSHR quotas destroys fairness (sub-figures (a) and (b)) and degrades system performance in terms of harmonic mean of speedups (sub-figure (c)) as a result of unfair treatment of memory-intensive applications in some workloads. The large increase in average unfairness is mainly due to workloads that contain the application *art*. *Art* is a highly memory-intensive workload with high row-buffer locality. As such, as we described in Section 4.3.5, it can cause bank service denial for concurrently executing applications even when it is throttled down. Additionally, *art*'s performance is very sensitive to the number of MSHR entries at its disposal. As a result, it can get *over-throttled* as



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

Figure 4.14: Effects of different throttling mechanisms for FST

described above when MSHR quotas are employed for throttling. Figure 4.14 (d) shows that while the over-throttling problem that exists for the workloads including *art* does not result in an average loss of system throughput (Wspeedup) across all the workloads, it does have a large impact on system fairness and average system turnaround time (as shown by Hspeedup, sub-figure (c)). We conclude that BSDP is necessary for significant improvements to system fairness when MSHR quotas are employed.

2. Without BSDP, the combination of MSHR quota and frequency throttling perform worse than using MSHR quota alone. The reason for this is the *over-throttling* of memory-intensive benchmarks in the absence of BSDP. When both throttling mechanisms are employed, the negative effect of *over-throttling* dominates the average in our evaluated workloads. This leads to the combination of the two throttling mechanisms performing worse than MSHR alone in the absence of BSDP.

3. Using *MSHR quotas* is more effective than using frequency throttling alone when BSDP is employed. Using *MSHR quotas* together with BSDP achieves 97% of the performance improvement and 95% of the fairness improvement provided by FST. However, as table 4.6 shows, some applications are not significantly slowed down by small adjustments to their MSHR quota values even when running alone. This is because applications such as *sphinx3* and *omnetpp* do not make use of many MSHRs even when running alone as they do not have high degrees of

memory-level parallelism. For such memory-intensive applications with low MLP, applying MSHR quotas as the throttling mechanism reduces the request rates only at the smallest throttling levels (MSHR quotas of 1 or 2). Therefore, using the second throttling mechanism that reduces the frequency at which requests are sent to L2 provides better, fine-grained control of request injection rate.

We conclude that using all mechanisms of FST is better than each throttling mechanism alone in terms of both fairness and performance.

# of MSHRs	1	2	3	5	6	12	32	64	128
sphinx3 (IPC)	0.13	0.23	0.28	0.29	0.29	0.30	0.30	0.30	0.30
milc (IPC)	0.10	0.22	0.36	0.38	0.39	0.40	0.40	0.40	0.40
leslie3d (IPC)	0.06	0.13	0.21	0.24	0.26	0.32	0.36	0.36	0.36
lbm (IPC)	0.04	0.10	0.22	0.26	0.30	0.39	0.45	0.46	0.48

Table 4.6: Sensitivity of alone performance to # of MSHRs

4.5.4 Evaluation of System Software Support

Enforcing Thread Priorities: As explained in Section 4.3.4, FST can be configured by system software to assign different weights to different threads. As an example of how FST enforces thread weights, we ran four identical copies of the *GemsFDTD* benchmark on a 4-core system and assigned them *thread weights* of 1, 1, 4 and 8 (recall that a higher-weight thread is one the system software wants to prioritize). Figure 4.15 shows that with no fairness technique each copy of *GemsFDTD* has an almost identical slowdown as the baseline does not support thread weights and treats the applications identically in the shared memory system. However, FST prioritizes the applications proportionally to their weights, favoring applications with higher weight in the shared memory system. FST also slows down the two copies with the same weight by the same amount. We conclude that FST approximately enforces thread weights, thereby easing the development of system software which naturally expects a CMP to respect thread weights in the shared memory system.

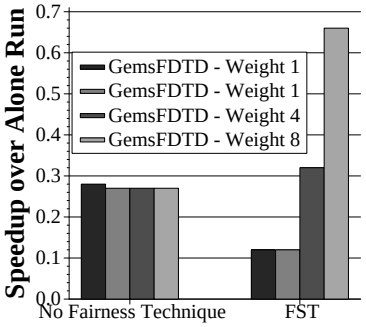


Figure 4.15: Enforcing thread weights with FST

Enforcing an Alternative Fairness Objective (Maximum Tolerable Slowdown): Section 4.3.4 explained how FST can be configured to achieve a *maximum slowdown threshold* as determined by system software, that dictates the maximum tolerable slowdown of any individual application executing concurrently on the CMP. Figure 4.16 shows an example of how FST enforces this fairness objective when four applications are run together on a 4-core system. The figure shows each application's individual slowdown in four different experiments where each experiment uses a different maximum slowdown threshold (ranging from 2 to 3) as set by the system software. As tighter goals are set by the system software, FST throttles the applications accordingly to achieve (close to) the desired maximum slowdown. The fairness objective is met until the maximum slowdown threshold becomes too tight and is violated (for *mgrid and parser*), which happens at threshold value 2. We conclude that FST can enforce different system-software-determined fairness objectives.

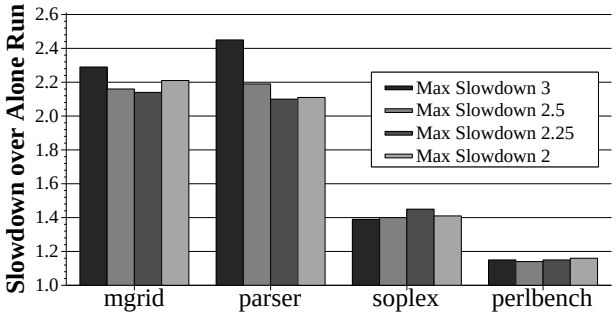
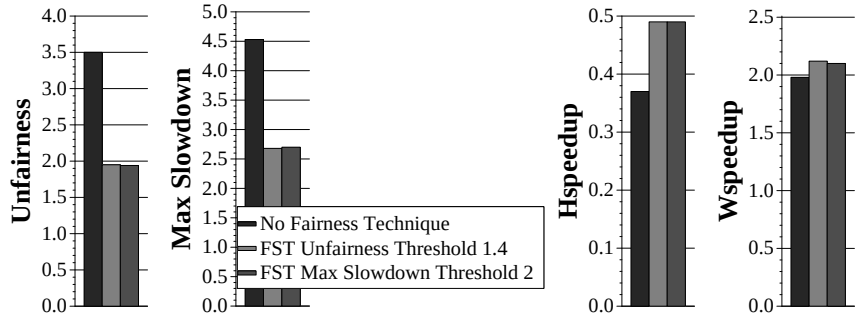


Figure 4.16: Enforcing maximum slowdown with FST

In Algorithm 3 throttling is triggered when estimated system unfairness is greater than a system-software-specified threshold. Figure 4.17 shows average system performance and fairness when using a system-software-specified maximum slowdown target compared to FST with an unfairness target which is the system-software target we use in all other experiments in this chapter. We conclude that similar system performance and fairness benefits can be gained using either system software goal: maximum tolerable slowdown or maximum tolerable unfairness.



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

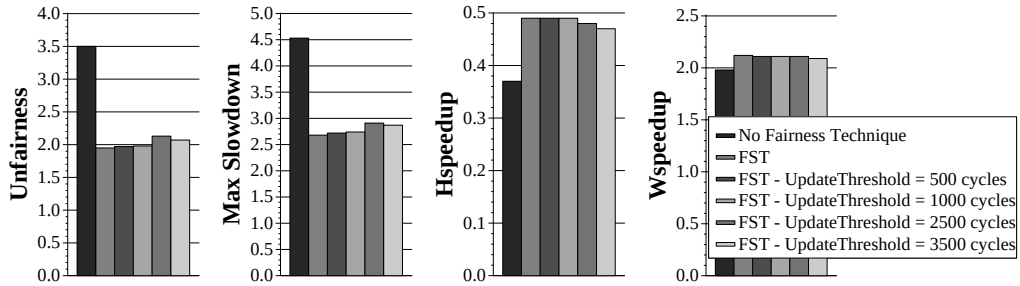
Figure 4.17: Comparing overall results with different system level targets

4.5.5 Effects of Implementation Constraints

Shared resources may be located far away from each other on the chip. In order to eliminate timing constraints on the sending of updates to the $InterferencePerCore$ bit-vector from the shared resources, such updates can be made periodically. Every $UpdateThreshold$ cycles, all shared resources send their local copies of $InterferencePerCore$ to update the main copy at the L2. Once the updates are applied to the main copy by taking the union of all bit-vectors, FST checks the main copy of $InterferencePerCore$. If the $InterferencePerCore$ bit of a core is set, FST increments the $ExcessCycles$ counter corresponding to the core by the $UpdateThreshold$ value.

Figure 4.18 shows the effect of periodic updates and sensitivity to chosen period lengths on the performance and fairness improvements of FST. The figure shows that even with updates occurring once every 1000 cycles, system perfor-

mance is almost identical and fairness improvements are within 2.5% of FST with updates being made every cycle. We conclude that using periodic updates (even when made at relatively long periods) eliminates any timing constraints on the sending of updates to the $InterferencePerCore$ bit-vector and does not significantly effect the performance and fairness improvements of FST.



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

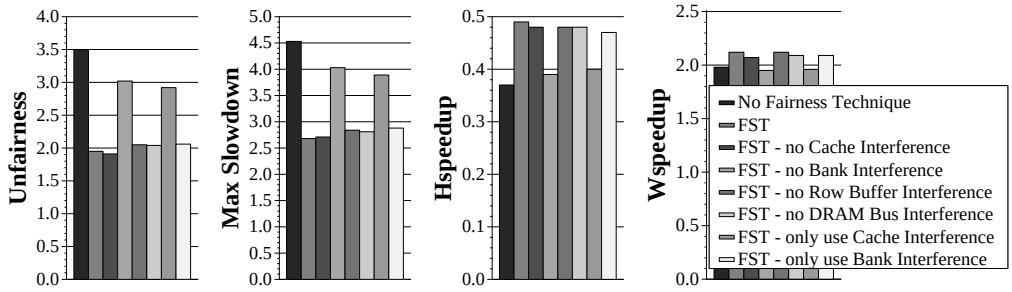
Figure 4.18: Effect of periodic updates on FST's performance and unfairness

4.5.6 Effects of Different Sources of Interference

Figure 4.19 shows the effect of taking into account interference from each of the interference sources we discuss in Section 4.3.3. The figure shows that FST's performance is mostly sensitive to whether or not DRAM bank interference is included in the estimations. Without DRAM bank interference, FST only improves performance by 5.1% (Hspeedup) and reduces unfairness by 13.8% respectively. As we observed in Section 4.3.6, a significant portion of the hardware required to implement FST is required for accounting for cache interference and DRAM row-buffer interference. This gives opportunity for a much less expensive implementation of FST based only on DRAM bank interference which can achieve 97% of the total performance improvements of FST and 94% of its total unfairness reduction.

4.5.7 Evaluation of Lightweight FST

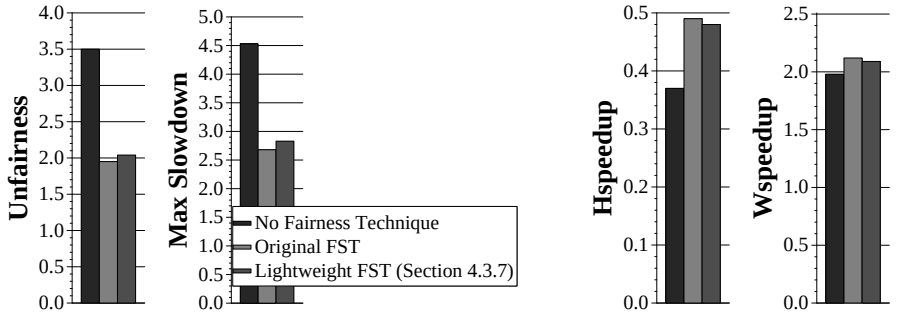
Figure 4.20 compares the performance of the lightweight FST implementation described in Section 4.3.7 to that of the baseline and the FST we have been



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

Figure 4.19: Sensitivity of FST to taking into account different interference sources

evaluating so far. The figure shows that the lightweight implementation that requires $2N$ counters for tracking *ExcessCycles* information, provides 98% of the system performance and 95% of the system fairness benefits of the original FST which requires N^2 counters. We conclude that this is an interesting option to consider for systems with a larger number of cores.



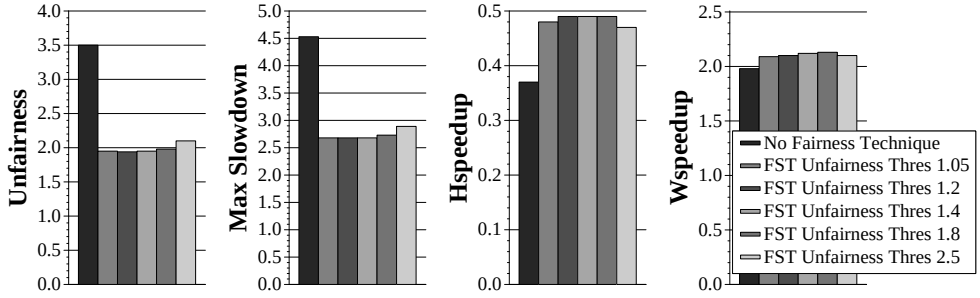
(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

Figure 4.20: Comparing overall results of original and lightweight FST

4.5.8 Sensitivity to Unfairness Threshold

Figure 4.21 shows how FST's average fairness and performance changes with different unfairness thresholds on our evaluated 4-core workloads. Lowering the *unfairness threshold* set by the system-software continuously improves fairness and performance until the unfairness threshold becomes too small. With a very small unfairness threshold (1.05), FST becomes 1) very aggressive at throttling down cores to reach the very tight unfairness goal, 2) too sensitive to inaccura-

cies in slowdown estimation and therefore triggers throttling of sources unnecessarily. As a result, both system performance and fairness slightly degrade. On the other hand, as the threshold increases, unfairness in the system also increases (because throttling is employed less often) and performance decreases beyond some point (because memory hog applications start causing starvation to others, leading to lower system utilization). Overall, the unfairness threshold provides a knob to the system software, using which the system software can determine the fairness-performance balance in the system. We find an unfairness threshold of 1.4 provides the best fairness and performance for our 4-core workloads.



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

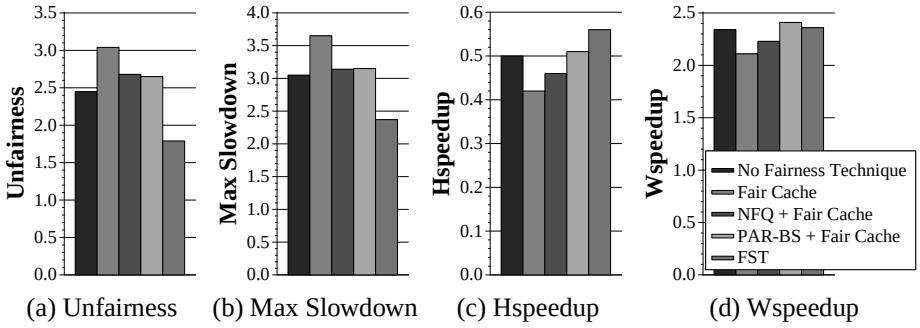
Figure 4.21: Sensitivity of FST to unfairness threshold

4.5.9 Effect of Multiple Memory Controllers

Figure 4.22 shows the effect of using FST on a system with two memory controllers. We conclude that in such a system with higher available off-chip bandwidth where there is less inter-core interference, and as a result lower unfairness to begin with in the baseline, FST provides significant improvements in system fairness and performance compared to combinations of fairness mechanisms at the different resources.

4.5.10 Evaluation of Using Profile Information

Figure 4.23 shows the effect of using profile information to account for slowdown due to throttling as described in Section 4.3.3.4. On average, using



(a) Unfairness (b) Max Slowdown (c) Hspeedup (d) Wspeedup

Figure 4.22: Effect of FST on a system with two memory controllers

such profile information improves system performance by 4% and leaves system unfairness unchanged across the 4-core workloads. However, such profile information is not completely accurate in accounting for slowdowns due to throttling in all intervals since the factors described in Section 4.3.3.4 are obtained by comparing performance of complete runs of each application at different throttling levels. Due to the inaccuracies that exist, the use of this information results in increased system unfairness in two of the workloads.

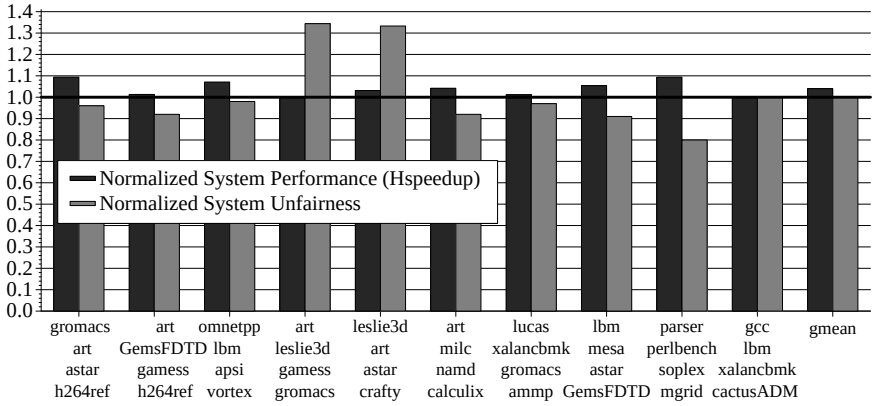


Figure 4.23: Effect of using profile information for throttling related slowdown

4.6 Conclusion

We proposed a low-cost architectural technique, Fairness via Source Throttling (FST), that allows system-software fairness policies to be achieved in CMPs by enabling fair sharing of the entire memory system. FST eliminates the need for

and complexity of multiple complicated, specialized, and possibly contradictory fairness techniques for different shared resources. The key idea of our solution is to gather dynamic feedback information about the slowdowns experienced by different applications in hardware at run-time and, based on this feedback, collectively adjust the memory request rates of sources (i.e., cores) to balance applications' slowdowns. Our solution ensures that fairness decisions in the entire memory system are made in tandem, thereby significantly improving both system performance and fairness compared to the state-of-the-art *resource-based* fairness techniques implemented independently for different shared resources. We have also shown FST is configurable by system software, allowing it to enforce thread priorities and achieve different fairness objectives. We conclude that FST provides a promising low-cost substrate that can not only improve the performance and fairness of future multi-core systems but also ease the task of future multi-core system software in managing shared on-chip hardware resources.

Chapter 5

Prefetch-Aware Shared-Resource Management

5.1 Introduction

In Chapter 4 we discussed how memory requests from different applications concurrently executing on different cores of a CMP interfere with one another. This inter-application interference causes each application to slow down compared to when it runs in isolation. Recent research (e.g., [57, 55, 15]) has proposed different mechanisms to manage this interference in the shared memory resources in order to improve system performance and/or system fairness. In Chapter 4, we proposed Fairness via Source Throttling (FST), which is one such technique targeted at providing fairness across all shared memory resources while providing high system performance.

On the other hand, memory latency tolerance mechanisms are critical to improving system performance as DRAM speed continues to lag processor speed. Prefetching is one commonly-employed mechanism that predicts the memory addresses a program will require, and issues memory requests to those addresses before the program needs the data. Prefetching improves the standalone performance of many applications and is currently done in almost all commercial processors [70, 27, 40, 61]. In Chapter 3, we proposed Hierarchical Prefetcher Aggressiveness Control (HPAC), which intelligently adjusts prefetcher aggressiveness at runtime to make prefetching effective and efficient in CMPs.

Ideally we would like CMP systems to both obtain the performance benefits of prefetching when possible, and also reap the performance and fairness benefits of shared resource management techniques. However, shared resource management techniques that otherwise improve system performance and fairness significantly,

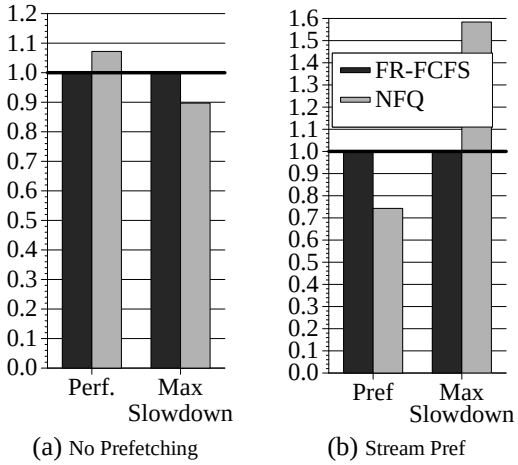
can also significantly degrade performance/fairness in the presence of prefetching. The reason: these techniques are designed for demand requests and do not consider prefetching.

Figure 5.1 illustrates this problem on a system that uses a fair/quality of service (QoS)-capable memory scheduler, network fair queuing (NFQ) scheduler [57]. Results are averaged over 15 multiprogrammed SPEC CPU2006 workloads on a 4-core system[1], and normalized to a system that uses a common first-ready first-come-first-serve (FR-FCFS) memory scheduler [65]. Figure 5.1 (a) shows how NFQ affects average system performance and average maximum slowdown (one metric of unfairness) in a system with no prefetching. Figure 5.1 (b) shows this in the presence of aggressive stream prefetching. This figure shows that, even though NFQ improves performance and reduces maximum slowdown on a system that does not have a prefetcher, if aggressive prefetching is enabled, we see a very different result. On a system with prefetching NFQ degrades performance by 25% while significantly increasing maximum slowdown, because its underlying prioritization algorithm does not differentiate between prefetch and demand requests. As a result, prefetches can be unduly prioritized by the memory scheduler, causing system performance and fairness degradation.

In this chapter, we demonstrate that different shared resource management techniques suffer from this problem, i.e., they can degrade performance significantly when employed with prefetching. Our goal is to devise general mechanisms that intelligently take prefetches into account within shared resource management techniques to ensure their effectiveness for both performance and fairness in the presence of prefetching.

We provide mechanisms for management of prefetch requests in three recently proposed shared resource management techniques. Two of these techniques are *resource-based* memory scheduling techniques: network fair queuing

[1]Our system configuration, metrics, and workloads are discussed in section 5.5. In Figure 5.1, the stream prefetcher of Table 5.1 is used. Prefetch and demand requests are treated alike with respect to NFQ's virtual finish time calculations.



(a) No Prefetching (b) Stream Pref

Figure 5.1: Harmonic mean of speedups and maximum slowdown on system using NFQ memory scheduler (normalized to FR-FCFS)

(NFQ) [57] and parallelism-aware batch scheduling (PARBS) [55]. The third technique is the *source throttling-based* technique for coordinated management of multiple shared resources (FST) which we proposed in Chapter 4.

Basic Ideas: Our mechanisms build upon three fundamental ideas. First, we use accuracy feedback from the prefetchers to decide how prefetch requests should be handled in each of the *resource-based* techniques. The key idea is to *not* treat all prefetches the same. An application's prefetches should be treated similar to the demand requests *only when* they are useful.

Second, treating useful prefetches like demands can significantly delay demand requests of memory non-intensive applications because such requests can get stuck behind accurate prefetches (and demands) of memory-intensive applications. This degrades system performance and fairness. To solve this problem, we introduce the idea of *demand boosting*: the key idea is to boost the priority of the demand requests of memory non-intensive applications over requests of other applications.

Third, with *source throttling-based resource management*, we observe that uncoordinated core and prefetcher throttling can cause performance/fairness degradation because throttling decisions for cores can contradict those for prefetchers. To solve this problem, we propose mechanisms that coordinate core and prefetcher

throttling based on interference feedback that indicates which cores are being unfairly slowed down.

5.2 Summary from Previous Chapters and Background

In Sections 2.2.1 and 2.2.2 we gave an overview of two of the shared resource management techniques that we discuss in this chapter, NFQ [57] and PARBS [55]. Here we briefly summarize FST from Chapter 4 and our prefetcher control technique, HPAC, from Chapter 3. We first briefly describe what we mean by system fairness in the presence of prefetching.

5.2.1 Fairness in the Presence of Prefetching

We evaluate fairness of a multi-core system executing a multi-programmed workload using the $MaxSlowdown$ metric as defined in Section 4.4.1. Recall from Section 3.4.1 that the *Individual Slowdown (IS)* of each application is calculated as T_{shared}/T_{alone}, where T_{shared} is the number of cycles it takes an application to run simultaneously with other applications, and T_{alone} is the number of cycles it would have taken the application to run alone on the same system. In all of our evaluations, we use an aggressive stream prefetcher when calculating each benchmark's T_{alone} as our stream prefetcher significantly improves average performance and makes for a better baseline system. In addition to the $MaxSlowdown$ metric, we also show the commonly used *unfairness* metric [36, 22, 54] as defined in Section 3.4.1.

5.2.2 Hierarchical Prefetcher Aggressiveness Control (HPAC)

In Chapter 3, we proposed hierarchical prefetcher aggressiveness control (HPAC) as a prefetcher throttling solution to improve prefetching performance in CMPs. HPAC's goal is to control/reduce inter-thread interference caused by prefetchers. It does so by gathering global feedback information about the effect of each core's prefetcher on concurrently executing applications. Examples of global feedback are memory bandwidth consumption of each core, how much each core is

delayed waiting for other applications to be serviced by DRAM, and cache pollution caused by each core's prefetcher for other applications in the shared cache. Using this feedback, HPAC throttles each core's prefetcher. By doing so, we showed that HPAC can enable system performance improvements using prefetching that are not possible without it. In this chapter, we use HPAC in our baseline system since it significantly improves the performance of prefetching in multi-core systems and therefore constitutes a stronger baseline.

5.2.3 Fairness via Source Throttling (FST)

In Chapter 4 we proposed fairness via source throttling (FST) as a mechanism to provide fairness in the entire shared memory system. FST dynamically estimates how much each application *i* is slowed down due to inter-core interference that results from sharing the memory system with other applications. Using these estimated slowdowns, FST calculates an estimate for system unfairness. In addition, FST also determines the core experiencing the largest slowdown in the system, referred to as *App-slowest*, and the core creating the most interference for *App-slowest*, referred to as *App-interfering*. If the estimated unfairness is greater than a threshold specified by system software, FST throttles down *App-interfering* (i.e., it reduces how aggressively that application accesses the shared memory resources), and throttles up *App-slowest*. In order to throttle down the interfering thread, FST limits the number of requests that the thread can simultaneously send to the shared resources and also the frequency at which it does so.

In order to estimate each application's slowdown, FST tracks inter-thread interference in the memory system. FST estimates *both* how much each application *i* is actually being slowed down due to inter-core interference and *also* how much each other core *j* ($j \neq i$) contributes to the interference experienced by core *i*. In Chapter 4, we assume all requests are demand requests and do not consider prefetching. In this chapter we demonstrate what problems occur when prefetching is enabled and extend the FST design of Chapter 4 to continue to be efficient in the presence of prefetching.

5.3 Motivation

In this section, we motivate why special treatment of prefetch requests is required in shared resource management techniques to both 1) achieve benefits from prefetching and, 2) maintain the corresponding techniques' performance benefits and/or fairness/QoS capabilities.

Every shared resource management technique has a prioritization algorithm that determines the order in which requests are serviced. For example, NFQ prioritizes requests that have earlier *virtual finish times*. PARBS prioritizes requests included in the formed batch by scheduling them all before a new batch is formed. In resource-based management techniques, the first key idea of this chapter is that the usefulness of prefetch requests should be considered within each management technique's prioritization policy. As such, not all prefetches should be treated the same as demand requests, and not all prefetches should be deprioritized compared to demand requests. However, this is not enough; in fact, prioritizing accurate prefetches causes starvation to demands of non-intensive applications. To solve this problem, the second key idea we present in this chapter is to boost the priority of demand requests of such non-intensive applications so that they are not starved.

We motivate these two key ideas with two examples.

Example 1: Figure 5.2 shows the effect of prefetching on PARBS. The figure shows a snapshot of the memory request buffers in the memory controller for banks 1 and 2. The initial state of these queues right before a new batch is formed can be seen on the left. Based on PARBS's batching algorithm, a maximum number of requests from any given thread to any given bank are marked to form a batch. Let us assume PARBS marks three requests per-thread per-bank when forming a batch. Additionally, let us assume that application 1's prefetches are useless or inaccurate while application 2's prefetches are useful or accurate. Figure 5.2 shows two simplistic policies, (a) and (b), and our proposed approach, policy (c), for handling prefetches in PARBS's batching phase. Figure 5.3 shows the respective memory service timelines.

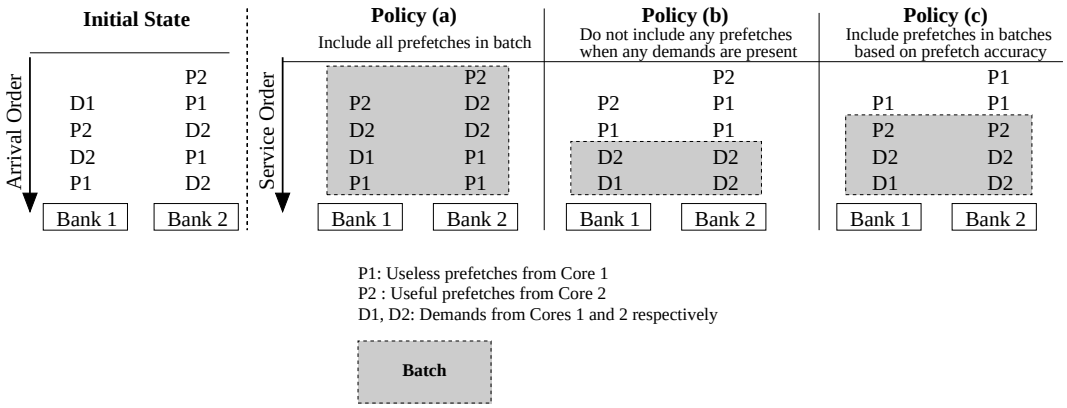


Figure 5.2: Example 1 - Different policies for treatment of prefetches in PARBS batch formation

Policy (a): mark prefetches and demands from each thread alike when creating a batch. Figure 5.2 shows that all the requests in the memory request queues of the two banks are included in the batch with this policy. Within each batch, PARBS prioritizes threads that that are "shorter jobs" in terms of memory request queue length. Since thread 1 has a shorter queue length (maximum 2 requests in any bank) than thread 2 (maximum 3 requests in any bank), thread 1 is prioritized over thread 2. As a result, as Figure 5.3 (a) shows, thread 1's inaccurate prefetches to addresses Y, X and Z are prioritized over thread 2's demands and useful prefetches. This leads to unwarranted degradation of thread 2's performance without any benefit to thread 1 (as its prefetches are useless).

Policy (b): never mark prefetches. This policy provides a naive solution to policy (a)'s problems by not marking any prefetches. This is helpful in prioritizing the demands of thread 2 over the useless prefetches of thread 1. However, by not marking any prefetches, this policy also does not include the useful prefetches of thread 2 in the generated batch. Figure 5.3 (b) shows that thread 2's useful prefetches to addresses L and M are now delayed since all prefetches are deprioritized. Hence thread 2 issues demands for addresses L and M before the prefetches are serviced, and so the benefit of those accurate prefetches significantly decreases. This causes a loss of potential performance.

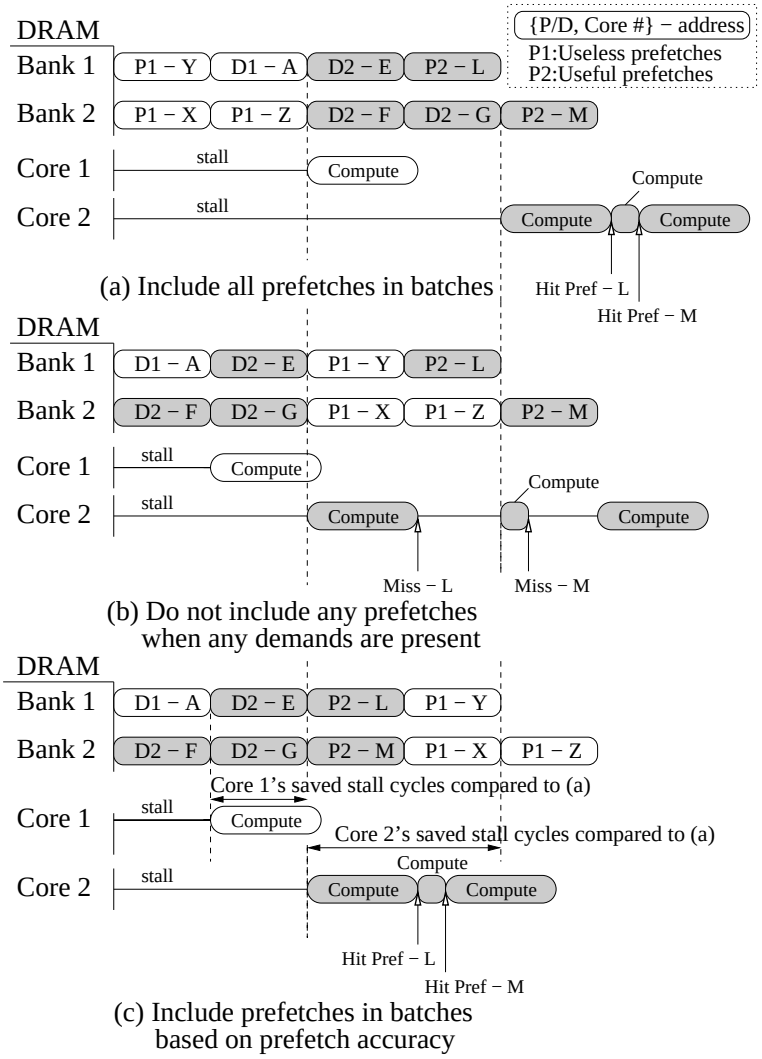


Figure 5.3: Memory service timeline for requests of Figure 5.2

Our Approach: A key principle we introduce in this chapter is to treat only accurate prefetches as demands in shared resource management. Figure 5.2 (c) concisely shows how this is done for PARBS. Using feedback from different threads' prefetchers, PARBS can make a more intelligent decision about whether or not to include prefetches when forming batches. Since thread 2's prefetches are useful, we include them in the batch, while thread 1's useless prefetches are excluded. As a result, benefits from prefetching for thread 2 is maintained, as shown in Figure 5.3 (c). Excluding thread 1's useless prefetches from the batch improves system fairness as

these requests do not unduly delay thread 2's demands and useful prefetches, and thread 2's slowdown is reduced without increasing thread 1's slowdown. Figure 5.3 (c) shows that this policy improves both applications' performance compared to policies that treat all prefetches equally. This motivates the need for distinguishing between accurate and inaccurate prefetches in shared resource management.

Example 2: Figure 5.4 shows the problem with just prioritizing accurate prefetches, and concisely shows our solution for a system using PARBS. When including accurate prefetches into the batches formed by PARBS, in the presence of prefetch-friendly applications (like application 2 in Figure 5.4), the size of the batches can increase. Since memory non-intensive applications (like application 1 in Figure 5.4) generate memory requests at a slow pace, every time a batch is formed (Time *t1* shown in Figure 5.4(a)), memory non-intensive applications will have a small number of their requests included. At time *t2*, more requests from the memory non-intensive application arrive. Without our proposed mechanism, since the current batch is still being serviced, these requests have to wait until the current batch is finished (Figure 5.4 (c)), which could take a long time since useful prefetch requests that were included in the batch made the batch size larger. In this chapter, we propose demand boosting, which prioritizes a *small number* of the non-intensive application's requests over others. In Figure 5.4 (d), at time *t3*, the two demand requests from application 1 to addresses K and L are boosted *into* the current batch and prioritized over the existing requests from application 2 within the batch. This allows application 1 to go back to its compute phase quickly. Doing

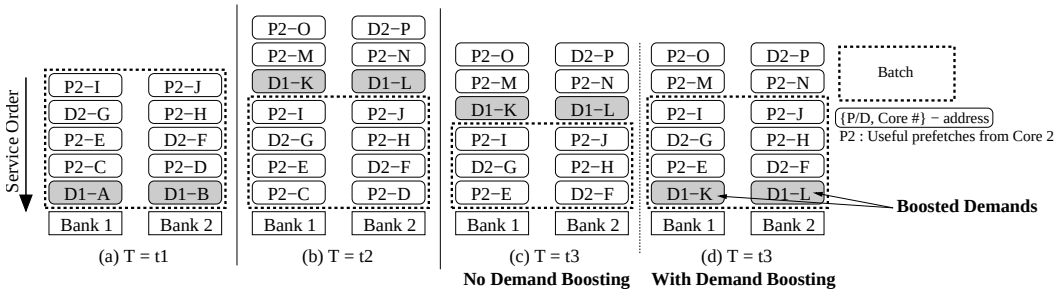


Figure 5.4: Example 2 - No demand boosting vs. Demand boosting

so does not degrade application 2's performance significantly as the non-intensive application 1 inherently has very few requests.

5.4 High Performance and Fair Shared Resource Management in the Presence of Prefetching

In this section, we describe in detail our proposal for handling prefetches in the two types of resource management techniques: *resource-based* and *source-based*. We also introduce *demand boosting*, which is orthogonal to the employed resource management technique. Since demand boosting is common to both resource-based and source-based techniques, we describe it first in Section 5.4.1. Then, we describe in detail how to apply our insights (described in Sections 5.1 and 5.3) to each resource management technique in turn (Sections 5.4.2 and 5.4.3).

5.4.1 Demand Boosting

Problem and Main Idea: As described in Section 5.3, the first component of our proposal is to treat useful prefetches to be as important as demands. Memory-intensive and prefetch-friendly applications can generate many such requests, which can cause long delays for the demands of concurrently executing memory non-intensive threads. As a result, system performance and fairness can degrade because of large performance degradations to memory non-intensive applications. To mitigate this problem, we propose *demand boosting* for such non-intensive applications. The key idea is to prioritize the non-intensive application's *small number of demand requests* over others, allowing that application to go back to its compute phase quickly. It must be noted that doing so does not significantly degrade other applications' performance because the non-intensive application inherently has very few requests.

Why the Problem Exists: The potential for *short-term* starvation of a non-intensive application's demands increases in each of the techniques we consider for different reasons. In NFQ and FST, potential for starvation is created by the priori-

tization of DRAM row buffer hits in the memory scheduler, coupled with high row buffer locality of accurate prefetches that are considered as important as demands. PARBS uses the batching concept to mitigate this inherent issue due to prioritizing row-buffer hit requests. However, in Section 5.3 we proposed including accurate prefetches into PARBS's batches. The slow rate at which non-intensive threads generate their requests, together with the large batches generated using requests from prefetch-friendly applications, causes potential for starvation in PARBS. In addition, when such memory non-intensive applications are cache friendly, as they stall waiting for their small number of memory requests to be serviced, their useful requests in the shared cache move up the LRU stack. Hence, they can get evicted more quickly by intensive applications' requests. This, in turn, causes larger performance penalties for such memory non-intensive applications.

To summarize, elevating the priority of accurate prefetch requests from memory intensive applications causes the small memory related stall times of non-intensive applications to increase. This significantly hurts the non-intensive applications' performance (as also observed by prior work [41]).

Demand Boosting Mechanism: Demand boosting is a general mechanism orthogonal to the type of resource management technique. It increases the performance of memory non-intensive applications that do not take advantage of accurate prefetches by dynamically prioritizing *a small number* of such applications' demands. With demand boosting, the demands of an application that does not have accurate prefetches ***and*** has a at most a *threshold number* of outstanding requests, will be boosted and prioritized over *all other* requests. For example, in a system using PARBS, when an application's demands are boosted, they no longer wait for a current batch to finish before they are considered for scheduling. A boosted request X has higher priority than any other request Y regardless of whether or not request Y is in the current batch.[2]

[2]Note that in the context of demand boosting for PARBS, demand boosting is significantly different from the "intra-batch" ranking proposed by the original PARBS mechanism (which we use in all our PARBS related mechanisms). PARBS's ranking prioritizes requests chosen from requests

Delaying a memory-intensive application in lieu of a memory non-intensive application with inherently small memory stall times can improve both system performance and fairness [55, 42, 15, 37]. In many cases, demand-boosting enables performance benefits from prefetching that are not possible without it, as we show in Section 6.4.

5.4.2 Prefetch-Aware Resource-Based Management Techniques

We identify prefetcher accuracy as the critical prefetcher characteristic to determine how a prefetcher's requests should be treated in shared resource management techniques. Prefetcher accuracy is defined as the ratio of useful prefetches generated by a prefetcher to the total number of prefetches it generates. We also investigated using other prefetcher feedback such as a prefetcher's *degree of timeliness*[3], but found that accuracy has more of a first order effect.

In all of the mechanisms we propose, we measure prefetch accuracy on an interval by interval basis. An interval ends when $T = 8192$ cache lines are evicted from the last level cache, where T is empirically determined. Every interval, information on the number of useful prefetches and total sent prefetches of each prefetcher is gathered. Using this information, the accuracy of the prefetcher in that interval is calculated and used as an estimate of the prefetcher accuracy in the following interval. In the following subsections, we discuss how to redesign underlying prioritization principles of two *resource-based* management techniques.

5.4.2.1 Parallelism-Aware Batch Scheduling

PARBS uses *batching* to provide a minimum amount of DRAM service to each application by limiting the maximum number of requests considered for

already contained *within the current batch* using its ranking algorithm. In contrast, with demand boosting, demand requests from a boosted thread are prioritized over *all* other requests.

[3]A prefetcher's degree of timeliness is defined as the ratio of the number of useful prefetches that fill the last level cache before the corresponding demand request is issued, to the total number of useful prefetches.

scheduling from any one application. Inaccurate prefetches of an application A can have negative impact on system performance and fairness in two ways. First, they get included in batches and get prioritized over other applications' demands and useful prefetches that were not included. As a result, they cause large performance degradation for those other applications without improving application A's performance. Second, they reduce the fairness provided by PARBS to application A by occupying a number of slots of each batch that would otherwise be used to give application A's demands a minimum amount of useful DRAM service.

We propose the following new batch scheduling algorithm to enable potential performance improvements from prefetching, while maintaining the benefits of PARBS. The key to Algorithm 4 is that it restricts the process of marking requests to demands and accurate prefetches. As a result, a prefetch-friendly application will be able to benefit from prefetching within its share of memory service. On the other hand, inaccurate requests are not marked and are hence deprioritized by PARBS.

Algorithm 4 Parallelism-Aware Batch Scheduler's Batch Formation (Prefetch-Aware PARBS, P-PARBS)

Forming a new batch: A new batch is formed when there are no marked requests left in the memory request buffer, i.e., when all requests from the previous batch have been completely serviced.
Marking: When forming a new batch, the scheduler marks up to *Marking-Cap* outstanding demand ***and also accurate prefetch requests*** for each application; these requests form the new batch.

5.4.2.2 Network Fair Queuing

NFQ uses *earliest virtual finish time first* memory scheduling to provide quality of service to concurrently executing applications. Inaccurate prefetches of some application A can have negative impact on system performance and fairness in two ways: First, if application A's inaccurate prefetches get prioritized over demands or accurate prefetches of some other application B due to the former's earlier virtual finish time, system performance will degrade. Application B's service is delayed while application A does not gain any performance. Second, since NFQ provides service to application A's inaccurate prefetches, the virtual finish times of

application A's demands grows larger than when there was no prefetching. This means that application A's demand requests will get serviced later compared to when there is no prefetching. Since application A's prefetches are not improving its performance, this ultimately results in application A's performance loss due to unwarranted waste of its share of main memory bandwidth.

We propose the following prioritization policy for the NFQ bank scheduler. When this scheduler prioritizes requests based on earliest virtual finish time, this prioritization is performed only for demand accesses and *accurate* prefetches. Doing so prevents the two problems described in the previous paragraph. Algorithm 5 summarizes the proposed NFQ policy.

Algorithm 5 Network Fair Queuing's Bank Scheduler Priority Policy (Prefetch-Aware NFQ, P-NFQ)

Prioritize ready commands (highest)
Prioritize CAS commands
Prioritize commands for demands ***and also accurate prefetch requests*** with earliest virtual finish-time
Prioritize commands based on arrival time (lowest)

5.4.3 Prefetch-Aware Source-Based Management Techniques

We propose prefetch handling mechanisms for the *source-based* shared resource management approach (FST), which we proposed in Chapter 4. We briefly described FST's operation in section 5.2.3. FST does not take into account *interference generated for prefetches* and *interference generated by the prefetches* of each application.

We incorporate prefetch awareness into FST in two major ways by: a) determining how prefetches and demands should be considered in estimating slowdown values, and b) coordinating core and prefetcher throttling using FST's monitoring mechanisms.

5.4.3.1 Determining Application Slowdown in the Presence of Prefetching

FST tracks interference in the shared memory system to dynamically estimate the slowdown experienced by each application. Yet, it cannot compute accurate slowdown values if prefetching is employed because FST is unaware of prefetches. In this section we describe a new mechanism to compute slowdown when prefetching is employed.

When requests A and B from two applications interfere with each other in a shared resource, one request receives service first and the other is *interfered-with*. Let us assume that request A was the *interfering* and request B was the *interfered-with*. The *type* of memory request A classifies the interference as *prefetch-caused* or *demand-caused* interference. The type of memory request B classifies the interference as *prefetch-delaying* or *demand-delaying* interference.

Recall that FST defines individual slowdown, *IS*, as T_{shared}/T_{alone} to estimate system unfairness. In order to estimate T_{alone} when running in shared mode, FST estimates "the number of *extra cycles* it takes an application to execute due to inter-core interference in the shared memory resources." This is known as T_{excess} ($T_{excess} = T_{shared} - T_{alone}$).

When estimating T_{excess} in the presence of prefetching, we find that it is important to use the following two principles. First, both *prefetch-caused* and *demand-caused* interference should be considered. Second, only *demand-delaying* interference should be used to calculate slowdown values at runtime. This means that when calculating core *i*'s T_{excess}, interference caused for its demands by *either* demands or prefetches of other cores *j* ($j \neq i$) should be accounted for. This is because ultimately both prefetch and demand requests from an interfering core can cause an *interfered-with* core to stall. On the other hand, even though *prefetch-delaying* interference reduces the timeliness of interfered-with prefetches, it does not significantly slow down the corresponding core. If an accurate prefetch is delayed until the corresponding demand is issued, that prefetch will be promoted to a demand. Further delaying of that request will contribute to the slowdown estimated

for the respective core because any interference with that request will be considered *demand-delaying* from that point on.

Algorithm 6 summarizes how our proposal handles prefetches to make FST prefetch-aware.[4] FST uses a bit per core to keep track of when each core was interfered with. We refer to this bit-vector as the *Interference* bit-vector in the algorithm. Also, an $ExcessCycles$ counter is simply used to track T_{excess} for each core.

Algorithm 6 Prefetch-aware FST (P-FST) estimation of T_{excess} for core *i*

Every cycle
if inter-core interference created by any core *j*'s ***prefetch requests*** or ***demand requests*** for core *i*'s ***demand requests*** **then**
  set core *i*'s bit in the $Interference$ bit-vector
end if
if Core *i*'s bit is set in the $Interference$ bit-vector **then**
  Increment $ExcessCycles$ counter for core *i*
end if

5.4.3.2 Coordinated Core and Prefetcher Throttling

FST throttles cores to improve fairness and system performance. On the other hand, HPAC is an independent technique that throttles prefetchers to improve system performance by controlling prefetcher-caused inter-core interference. Unfortunately, combining them without coordination causes contradictory decisions. For example, the most slowed down core's prefetcher can be throttled down (by the prefetch throttling engine, i.e., HPAC's global control) while the core is being throttled up (by the core throttling engine, i.e. FST). As a result, fairness and performance degrade and potential performance benefits from prefetching can be lost. Therefore, we would like to coordinate the decisions of core and prefetcher throttling. The key insight is to coordinate HPAC's throttling decisions with FST's decisions using the interference information collected by FST. We achieve this in two ways.

[4] We present our changes to the original T_{excess} estimation algorithm presented in Algorithm 2 of Chapter 4. For other details on T_{excess} estimation we refer to Section 4.3.3.

The first key idea is to use the slowdown information that FST gathers for core throttling to make better prefetcher throttling decisions. To do this, we only apply HPAC's global prefetcher throttle down decisions to a core if FST has detected the corresponding core to be $App_{interfering}$.[5] As such, we *filter* some of the throttle-down decisions made by HPAC. This is because HPAC can be very strict at prefetcher throttling due to its coarse classification of the severity of prefetcher-caused interference. As a result, it throttles some prefetchers down *conservatively* even though they are not affecting system performance/fairness adversely. We avoid this by using the information FST gathers about which cores are actually being treated unfairly as a result of inter-core interference.

The second key idea is to use FST's ability of tracking inter-core cache pollution to improve how well HPAC detects accurate prefetchers. This is useful because HPAC can underestimate a prefetcher's accuracy due to its interference-unaware tracking of useful prefetches. HPAC does not count accurate prefetches for core *i* that were evicted by some other core's requests before being used. This can cause HPAC to incorrectly throttle down core *i*'s accurate prefetcher and degrade its performance. To avoid this, we use FST's pollution filter to detect when an accurate prefetch for core *i* was evicted due to another core *j*'s request. For this purpose, we extend FST's pollution filter entries to also include a prefetch bit. Using this, we account for useful prefetches evicted by another core's requests in HPAC's estimate of each prefetcher's accuracy.

Algorithms 7 and 8 summarize the above mechanisms that coordinate core and prefetcher throttling.

[5] If HPAC's local throttling component for core *i* detects that the core's prefetcher is not performing well, that prefetcher is still throttled down regardless of FST's decision. This helps both core *i*'s and other cores' performance.

Algorithm 7 Prefetch-Aware FST (P-FST) Core and Prefetcher Throttling

if $Estimated\,Unfairness > Unfairness\,Threshold$ **then**
  Throttle down $App_{interfering}$
  Throttle down prefetcher of $App_{interfering}$ **if HPAC indicates global throttle down for this prefetcher**
  Throttle up $App_{slowest}$
end if
Allow HPAC to throttle up prefetchers as it requires
Apply HPAC's local throttle down decisions

Algorithm 8 Enhancing prefetcher accuracy information using FST's pollution filters

if Last-level cache hit on prefetched cache line **then**
  increment useful prefetch count
end if
if Last-level cache miss due to inter-core interference as detected by FST **and** evicted line was prefetch request **then**
  increment useful prefetch count
end if
Prefetch accuracy = useful prefetch count / total prefetch count

5.5 Methodology

5.5.1 Metrics

To measure CMP system performance, we use *Harmonic mean of speedups (Hspeedup)* [49], *Weighted speedup (Wspeedup)* [66], and *Individual speedup (IS)*, which are defined in Section 3.4.1. Since *Hspeedup* provides a balanced measure between fairness and system throughput [49], we use it as our primary evaluation metric. To demonstrate fairness improvements, we report *MaxSlowdown* (Section 4.4.1), and also *Unfairness* as defined in Section 3.4.1 (also see Section 5.2.1).

5.5.2 Processor Model

Table 5.1 shows the baseline configuration of each core and the shared resource configuration for the 4-core CMP system we use in the evaluations of this chapter. We faithfully model all port contention, queuing effects, bank conflicts, and other major DDR3 DRAM system constraints in the memory subsystem.

Execution core	15 stage out of order processor Decode/retire up to 4 instructions Issue/execute up to 8 micro instructions 128-entry reorder buffer
Front end	Fetch up to 2 branches; 4K-entry BTB 64K-entry Hybrid branch predictor
On-chip caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line L1 D-cache: 32KB, 4-way, 2-cycle, 64B line Shared unified L2: 2MB , 16-way, 16-bank, 20-cycle, 1 port, 64B line size
Prefetcher	Stream prefetcher with 32 streams, prefetch degree of 4, and prefetch distance of 64 cache lines [70, 67]
DRAM controller	On-chip, Open-row PARBS [55]/NFQ [57]/FR-FCFS [65] 128-entry MSHR and memory request queue
DRAM and bus	667MHz bus cycle, DDR3 1333MHz [50] 8B-wide data bus, 8 DRAM banks, 16KB row buffer per bank Latency: 15-15-15ns; 100-100-100 processor cycles (^{t}RP-^{t}RCD-CL), Round-trip L2 miss latency: Row-buffer hit: 36ns, conflict: 66ns

Table 5.1: Baseline system configuration

5.5.3 Workloads

We use the SPEC CPU 2000/2006 benchmarks for our evaluation. Each benchmark was compiled using ICC (Intel C Compiler) or IFORT (Intel Fortran Compiler) with the -O3 option. Each benchmark runs the reference input set for 50 million x86 instructions selected by Pinpoints [62].

We classify a benchmark as *memory-intensive* if its L2 Cache Misses per 1K Instructions (MPKI) is greater than three and otherwise we refer to it as *memory non-intensive.* We say a benchmark has *cache locality* if the number of L2 cache hits per 1K instructions for the benchmark is greater than five. An application is classified as *prefetch-friendly* if its IPC improvement due to prefetching when run in isolation is more than 10%. If its IPC degrades, it is classified as *prefetch-unfriendly* and otherwise as *prefetch-insensitive*. These classifications are based on measurements made when each benchmark was run alone on the 4-core system. Table 5.2 shows the characteristics of the 29 benchmarks that appear in the evaluated workloads when run on the 4-core system.

We used 15 four-application workloads for our evaluations. The workloads

were chosen such that each workload consists of at least two *memory-intensive* applications (MPKI greater than three) and an application *with cache locality*. All but one workload has at least one *prefetch-friendly* application since the goal of the chapter is to demonstrate how to improve system performance due to prefetching in systems that employ the different shared resource management mechanisms. The one workload with no prefetch-friendly applications consists of memory-intensive and prefetch-unfriendly applications.

		No prefetching		Prefetching				
Benchmark	Type	IPC	MPKI	IPC	MPKI	HPKI	Acc(%)	Cov(%)
art	FP00	0.23	25.7	0.25	13.73	105	61	55
gromacs	FP06	1.17	0.22	1.2	0.07	11	66	70
lbm	FP06	0.33	19.3	0.36	3.43	27.4	94	82
GemsFDTD	FP06	0.38	12.67	0.67	0.07	17.6	93	99
omnetpp	INT06	0.34	8.79	0.34	8.72	5	11	19
zeusmp	FP06	0.66	3.97	0.75	1.92	17	67	52
bzip2	INT06	1.57	0.96	1.65	0.64	7.8	95	35
perlbmk	INT00	1.8	0.04	1.8	0.03	5.4	16	35
xalancbmk	INT06	1.07	0.83	0.93	0.99	18.8	11	18
sphinx3	FP06	0.26	12.82	0.51	2.71	14.5	58	79
leslie3d	FP06	0.29	21.37	0.55	4.73	22.3	94	78
bwaves	FP06	0.26	22.43	0.33	2.3	11.3	100	90
astar	INT06	0.17	23.04	0.17	21.4	10.4	25	8
vortex	INT00	0.97	1.21	0.93	1.15	7	27	14
swim	FP00	0.39	16.85	0.48	0.57	20	100	97
h264ref	INT06	1.89	0.77	1.86	0.43	2	56	55
crafty	INT00	1.56	0.26	1.61	0.19	8	34	29
libquantum	INT06	0.26	11.84	0.29	2.21	0.52	100	81
applu	FP00	0.55	13.09	1.33	0.7	12.13	97	95
wrf	FP06	0.53	8.6	0.86	1.06	11.61	95	88
apsi	FP00	1.2	1.54	1.23	1.33	14.83	95	14
parser	INT00	1.11	0.68	1.21	0.12	8.25	78	82
gobmk	INT06	1.16	0.38	1.18	0.25	7.6	41	36
twolf	INT00	1.05	0.35	1.1	0.14	25.47	95	60
equake	FP00	0.27	18.72	0.4	3.54	8.77	98	81
mesa	FP00	1.58	1.96	1.58	1.92	0.89	61	2
gamess	FP06	2.04	0.15	2.12	0.04	4.64	58	75
lucas	FP00	0.47	10.42	0.61	4.8	5.1	99	54
ammp	FP00	1.92	0.33	1.93	0.29	14.64	9	13

Table 5.2: Characteristics of 29 SPEC 2000/2006 benchmarks that appear in the workloads of this chapter: IPC and MPKI (L2 cache Misses Per 1K Instructions) with and without prefetching, HPKI (L2 cache Hits Per 1K Instructions) with prefetching, and prefetcher accuracy and coverage

5.5.4 Parameters Used in Evaluation

In all our mechanisms, the threshold to determine whether an application's prefetcher is accurate is 80%. In P-NFQ and P-FST, an application must have *fewer than* ten memory requests in the memory request queue of the memory controller to be considered for *demand boosting*, and fewer than 14 requests in P-PARBS (section 5.6.5 shows that the reported results are not very sensitive to the value chosen for this threshold). The parameter setup for each of the FST and HPAC techniques is the same as those reported in [15] and [17] respectively. For PARBS [55], we use the same *Marking Cap* threshold as used in the original paper, five memory requests per thread per bank.

5.6 Experimental Evaluation

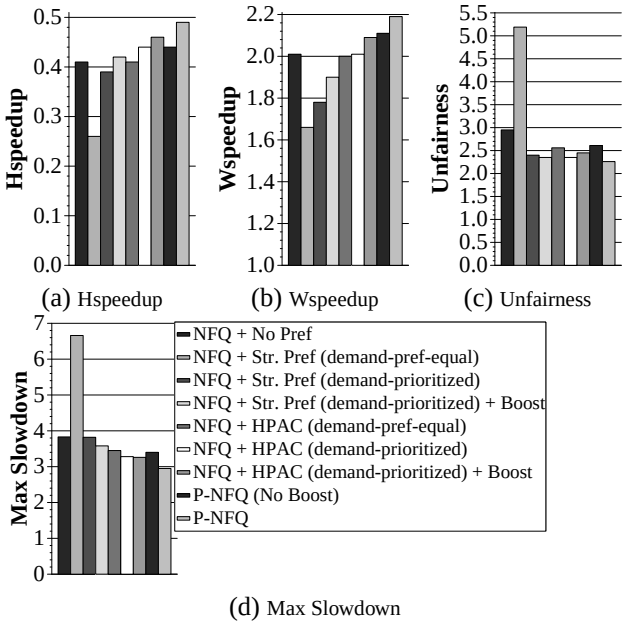
We evaluate the mechanisms described in the previous sections on a 4-core CMP system employing NFQ, PARBS, and FST in the following three subsections respectively. Note that our prefetch-aware NFQ, PARBS, and FST techniques (P-NFQ, P-PARBS, and P-FST) are evaluated on a system which includes the prefetcher aggressiveness control (HPAC) mechanism of Chapter 3.

5.6.1 NFQ Results

Figures 5.5 (a)-(d) show average system performance and unfairness of a system using an NFQ memory scheduler in different configurations: with no prefetching, prefetching with and without prefetcher control, and with our proposed prefetch-aware NFQ. In the policies referred to as *demand-pref-equal*, demands and prefetches are treated equally in terms of prioritization based on earliest virtual finish time. In the *demand-prioritized* policy, demands are always prioritized over prefetches, and are scheduled earliest virtual finish time first. Figure 5.6 shows system performance for each of the 15 evaluated workloads for the nine configurations of NFQ that we evaluated. P-NFQ provides the highest system performance and least unfairness among all the examined techniques. P-NFQ outperforms the best

performing previous technique (NFQ + HPAC demand-prioritized) by 11%/8.6% (HS/WS) while reducing maximum slowdown by 9.9%. Several key observations are in order:

1. Figure 5.5 shows that in all cases (with or without prefetcher throttling), *demand-prioritized* has higher performance and lower maximum slowdown than *demand-pref-equal*. We conclude that as we explained in section 5.4.2, if *all* prefetch requests are treated alike demand requests, wasted service given to useless prefetches leads to a worse-performing and less fair system than always prioritizing demands.



(a) Hspeedup (b) Wspeedup (c) Unfairness

(d) Max Slowdown

Figure 5.5: Average system performance and unfairness on 4-core system with NFQ

2. The last two bars in each of the subfigures of Figure 5.5 demonstrate **a key insight**: without intelligent prioritization of demand requests of memory non-intensive applications, system performance and fairness do not significantly improve *simply by* prioritizing accurate prefetches. Adding the demand boosting optimization to P-NFQ (with no boosting) improves performance by 10%/3.8% (HS/WS) and reduces maximum slowdown by 13.2% compared to just prioritizing accurate prefetches within NFQ's algorithm.

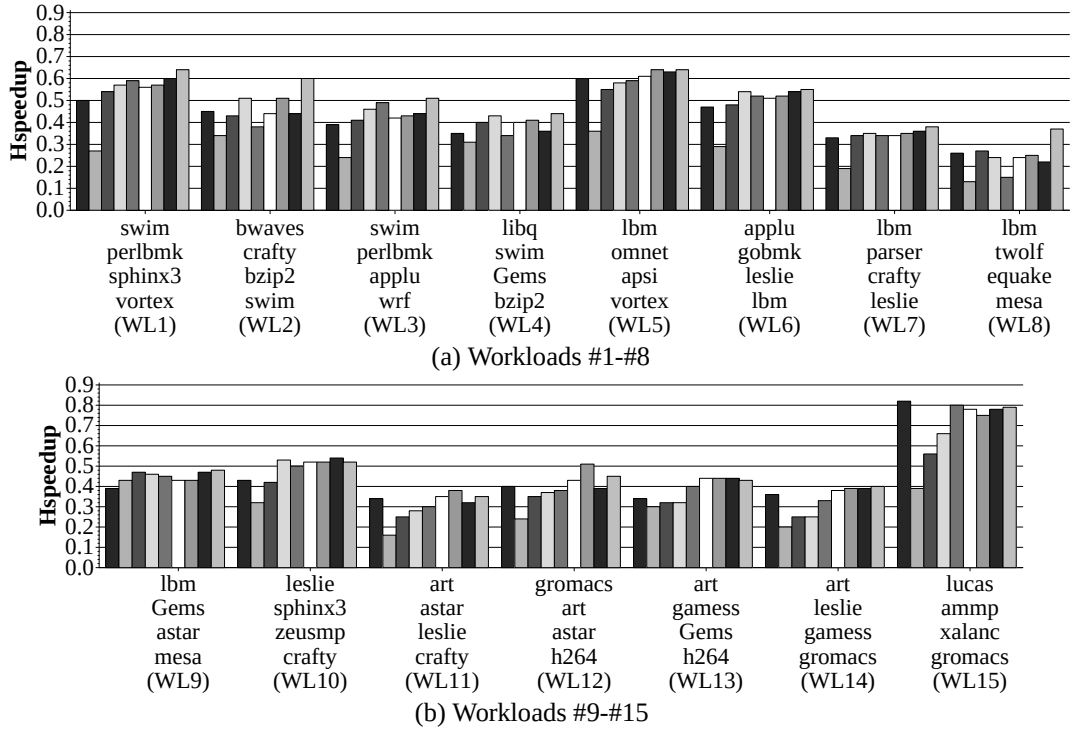
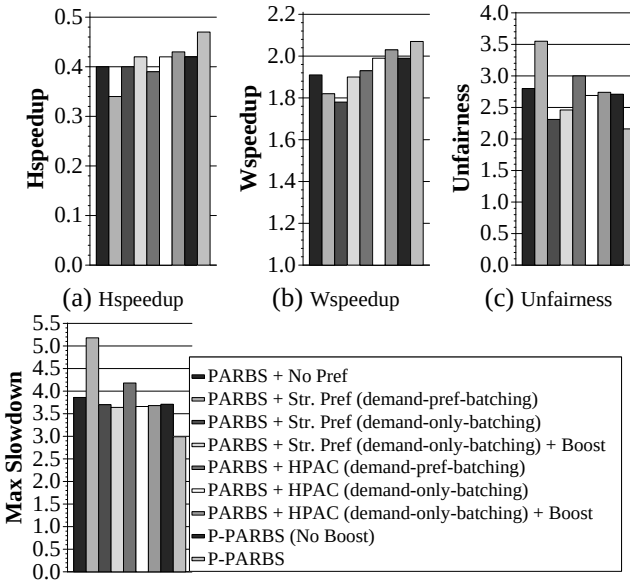


Figure 5.6: System performance (Hspeedup) for each of the 15 workloads with NFQ (legend same as Figure 5.5)

3. Figures 5.5 (a)-(d) show that demand boosting improves system performance independent of the setup it is used with. Demand boosting alone improves the performance of demand-prioritized and prefetching with no throttling by 7.3%/6.7% (HS/WS). When used with demand-prioritized and HPAC, it improves performance by 3.3%/3.6% (HS/WS). However, demand boosting provides the best system performance and fairness when used *together* with our proposed P-NFQ which prioritizes requests based on virtual finish time first using prefetch accuracy feedback. Note that demand boosting and considering prefetch accuracy information in prioritization decisions are synergistic techniques. Together they perform better than each one alone. We conclude that demand boosting is a general mechanism but is most effective when used together with resource management policies which take prefetcher accuracy into account in their prioritization rules.

5.6.2 PARBS Results

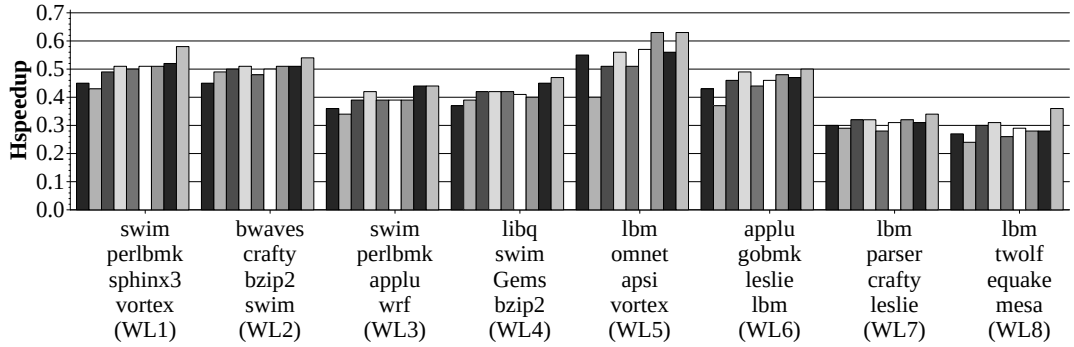
Figures 5.7 (a)-(d) show average system performance and unfairness of different prefetch-demand batching policies with and without prefetcher control. In *demand-pref-batching*, demands and prefetches are treated equally in PARBS's batch-forming (within the batches, demands are prioritized over prefetches because we find this to be better performing on average). In *demand-only-batching*, only demands are included in the batches. Figure 5.8 shows system performance for each of the 15 evaluated workloads for the nine configurations of PARBS that we evaluated. P-PARBS provides the highest system performance and the smallest unfairness among all of the techniques, improving system performance on average by 10.9%/4.4% (HS/WS) while reducing maximum slowdown by 18.4% compared to the combination of PARBS and HPAC with demand-only-batching.



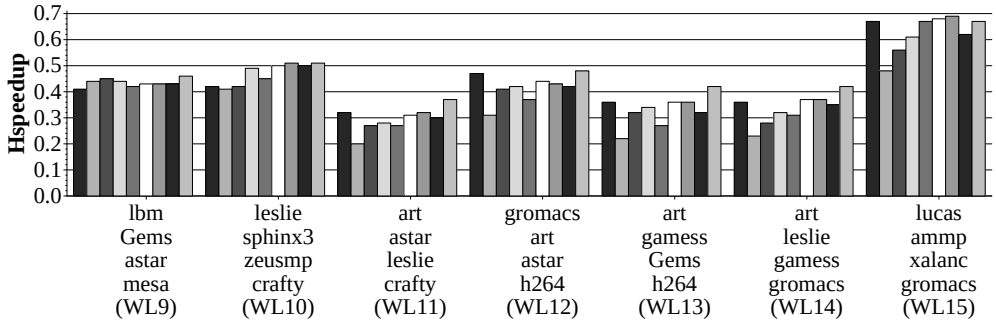
(a) Hspeedup (b) Wspeedup (c) Unfairness

(d) Max Slowdown

Figure 5.7: Average system performance and unfairness on 4-core system with PARBS



(a) Workloads #1-#8



(b) Workloads #9-#15

Figure 5.8: System performance (Hspeedup) for each of the 15 workloads with PARBS(legend same as Figure 5.7)

5.6.2.1 Case Study

The goal of this case study is to provide insight into how the mechanisms that we propose improve performance. It also shows in detail why *simply* prioritizing accurate prefetches in shared resource management techniques does not necessarily improve *system performance and fairness.* We examine a scenario where two memory intensive and prefetch-friendly applications (*swim and sphinx3*) concurrently execute with two memory non-intensive applications (*perlbmk and vortex*). Figures 5.9 (a) and (c)-(f) show individual application performance and overall system behavior of this workload. Figure 5.9 (b) shows the dynamics of the mechanisms proposed for prefetch-aware PARBS. In Figure 5.9 (b), each application is represented with two bars. The left bar in each pair shows the percentage of time that *both* demands and prefetches from the corresponding application were included in P-PARBS's batches vs. the percentage of time that *only* demands were included.

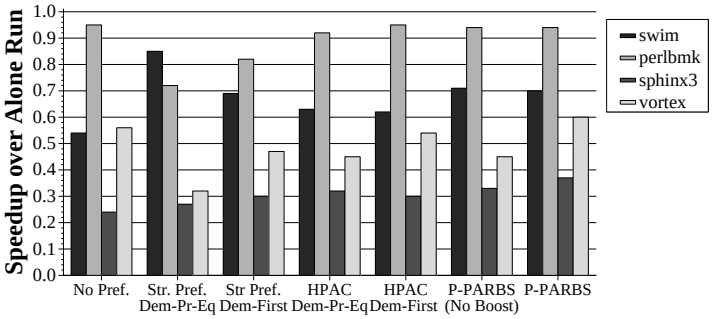
The right bar shows the percentage of all demand requests that were boosted into the batches by the demand-boosting mechanism vs. all other batched requests.

P-PARBS both performs significantly better and is much more fair than all the other evaluated techniques. This is due to the following two reasons:

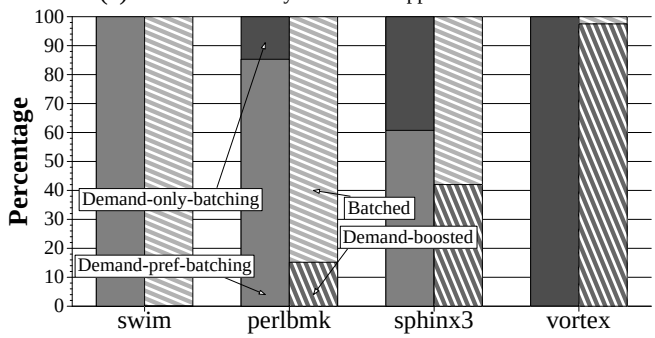
1. Including useful prefetches of *swim* and *sphinx3* alongside demand requests in P-PARBS's batches allows these applications to make good use of their accurate prefetches and significantly improves their performance. Figure 5.9 (b) shows that *swim*'s and *sphinx3*'s prefetches are included in the batches for 100% and 60% of their execution times respectively. During these periods, *swim* and *sphinx3* also achieve better row buffer locality: their row buffer hits are increased by 90% and 27% respectively compared to the technique with the best system performance among the other techniques (HPAC demand-only-batching). In addition, *swim* and *sphinx3*'s prefetches become 8% and 11% more timely (not shown in the figure).

2. Boosting the demands of the prefetch insensitive and memory non-intensive application, *vortex*, allows it to get quick memory service and prevents it being delayed by the many requests batched for *swim* and *sphinx3*. Because *vortex*'s requests are serviced quickly, its performance increases. Also, since *vortex* is memory non-intensive, this boosting does not degrade other applications' performance significantly.

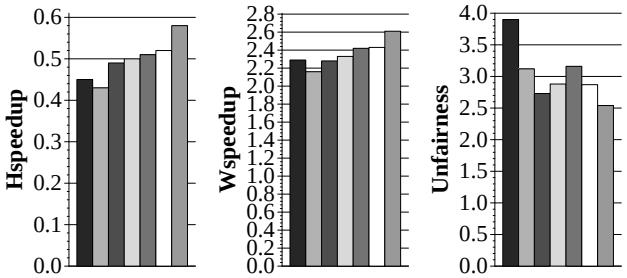
The last two sets of bars in Figure 5.9 (a) show the importance of demand boosting. When *swim*'s and *sphinx3*'s prefetches are included in the batches, *vortex*'s performance degrades if *demand boosting* is not used. This happens because of inter-core cache pollution caused by *swim* and *sphinx3*. Hence, even though *swim*'s and *sphinx3*'s performance improves significantly without boosting, overall system performance does not improve over the HPAC demand-only-batching (Figures 5.9 (c)-(d)). In contrast, with *demand boosting*, *vortex*'s performance also improves which enables P-PARBS to perform 13.3%/7.6% (HS/WS) better than the best previous approach while also reducing maximum slowdown by 17.8%.



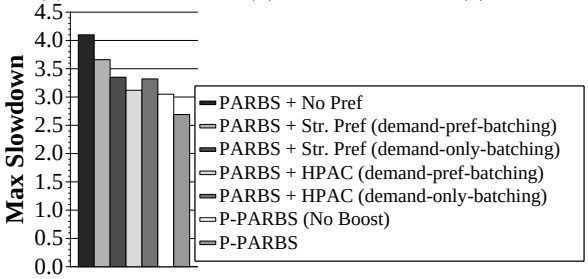
(a) PARBS case study: individual application behavior



(b) Left bars: dem-pref-batching vs dem-only-batching time, right bars: requests boosted vs batched normally



(c) Hspeedup (d) Wspeedup (e) Unfairness



(f) Max Slowdown

Figure 5.9: PARBS case study

5.6.3 FST Results

Figures 5.10 (a)-(d) show average system performance and unfairness of FST in the following configurations: without prefetching, with aggressive stream prefetching, with HPAC, and our proposed coordinated core and prefetcher throttling, i.e., P-FST (with and without demand boosting). Figure 5.11 shows system performance for each of the 15 evaluated workloads for the five configurations of FST that we evaluated. P-FST provides the highest performance and best fairness among the five techniques. Several observations are in order:

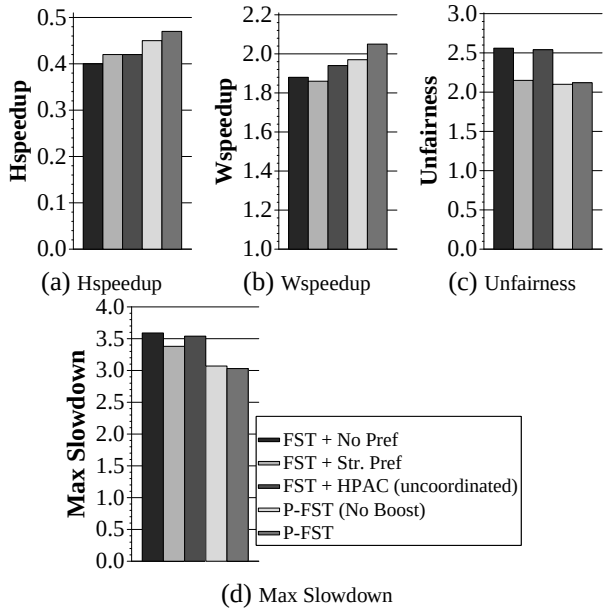


Figure 5.10: Average system performance and unfairness on 4-core system with FST

1. When prefetching with no throttling is used, in five of the workloads prefetcher-caused interference is noticeable and is left uncontrolled by FST. This results in large degradations in system performance of 5% or more (WL5, WL11, WL12, WL14, and WL15). In these workloads, FST does not detect the applications causing prefetcher interference to be *App-interfering*. Because of these workloads, prefetching with no throttling does not improve average system performance significantly compared to no prefetching as shown in Figure 5.10. This shows the need for explicit prefetcher throttling when prefetching is used with FST.

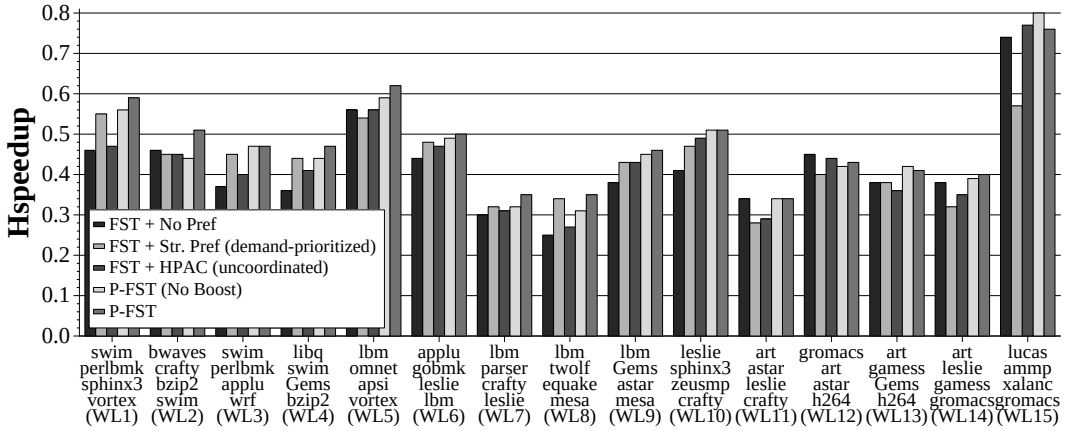


Figure 5.11: System performance (Hspeedup) for each of the 15 workloads

2. When HPAC (Chapter 3) and FST (Chapter 4) are naively combined with no coordination, four of the 15 workloads lose significant prefetching performance (workloads WL1, WL3, WL4, and WL8). In such cases, HPAC throttles down some useful prefetchers unnecessarily. This happens due to: a) excessive throttling caused by HPAC's coarse classification of interference, and b) underestimation of prefetcher accuracy due to interference-unaware tracking of useful prefetches (described in section 5.4.3.2). Unnecessary throttling makes the system more unfair compared to no prefetcher throttling. This happens when a prefetch-friendly application with the largest slowdown in the absence of prefetching is unnecessarily throttled. With no prefetcher throttling, such an application gains significant performance, which in turn reduces system unfairness. When HPAC throttles down the prefetchers of such applications too much, this fairness improvement is lost. We conclude that even though a naive combination of HPAC and FST improves average system throughput, this comes at the cost of increasing system unfairness significantly compared to no throttling.

3. Our P-FST technique (with demand-boosting) addresses the problems described above by coordinating prefetcher and core throttling, and improves performance by 11.3%/5.6% (HS/WS) while reducing maximum slowdown by 14.5% compared to the best performing of the other techniques (i.e. the uncoordinated FST and HPAC combination). Compared to the configuration with the least max-

imum slowdown, i.e. the combination of prefetching with no throttling and FST, P-FST with boosting performs 11.2%/10.3% (HS/WS) better while reducing maximum slowdown by 10.3%.

5.6.4 Effect on Homogeneous Workloads

Multi-core systems are sometimes used to run multiple copies of the same application in server environments. Table 5.3 shows system performance and fairness deltas of P-NFQ compared to NFQ + HPAC (demand-prioritized) for a prefetch friendly (four copies of sphinx3) and a prefetch unfriendly (four copies of astar) workload. Our proposal improves system performance and reduces max slowdown for the prefetch friendly workload, while it does not significantly affect the prefetch unfriendly one. In the prefetch friendly workload, prioritizing accurate prefetches improves each benchmark's performance by making timely use of those accurate prefetches. This is not possible if all prefetches are treated alike.

Four copies of sphinx3 (prefetch friendly)			Four copies of astar (prefetch unfriendly)		
Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown
7.9%	7.9%	-8.1%	-1%	-1%	0.5%

Table 5.3: Effect of our proposal on homogeneous workloads in system using NFQ memory scheduling

5.6.5 Sensitivity to System and Algorithm Parameters

Table 5.4 shows how P-NFQ performs compared to NFQ + HPAC (demand prioritized) on systems with two/four memory channels or 8MB/16MB shared last level caches. Even though using multiple memory channels reduces contention to DRAM, and using larger caches reduces cache contention, P-NFQ still performs significantly better while reducing maximum slowdown. We conclude that our mechanism provides performance benefits even on more costly systems with higher memory bandwidth or larger shared caches.

Figure 5.12 shows how sensitive the performance benefits of the techniques we propose (compared to the best previous technique in each case) are to the boost-

Single Channel			Dual Channel			Four Channel		
Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown
11%	8.6%	-9.9%	5%	5.7%	-3.7%	4%	6.3%	0.7%
2MB Shared Cache			8MB Shared Cache			16MB Shared Cache		
Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown	Δ HS	Δ WS	Δ Max Slowdown
11%	8.6%	-9.9%	6.3%	5.3%	-9.1%	4.9%	3.9%	-6.6%

Table 5.4: Effect of our proposal on system using NFQ memory scheduling with different microarchitectural parameters

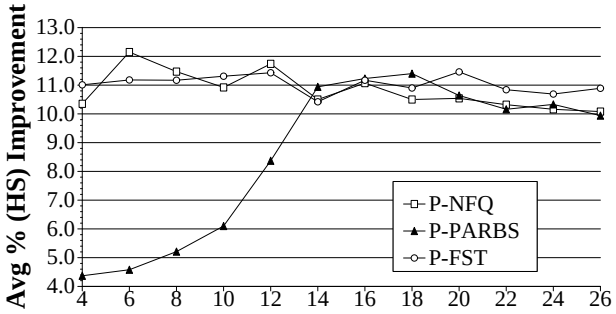


Figure 5.12: Sensitivity to boosting threshold

ing threshold. For all shown thresholds, P-NFQ and P-FST show performance within 1% of that of the chosen threshold. For P-PARBS, this is the case for all boosting threshold values between 14 and 26. In P-PARBS, with thresholds less than 14, not enough requests from prefetch-unfriendly benchmarks get boosted. We conclude that the benefits of our mechanisms are not highly sensitive to the chosen threshold value.

5.6.6 Hardware Cost

Table 5.5 shows the required storage of our mechanisms on top of each of the shared resource management techniques. Our mechanisms do not require any structures that are on the critical path of execution. Additionally, none of the structures we add/modify require large energy to access and none are accessed very often. As such, significant power overhead is not introduced.

P-NFQ	Closed form for N cores (bits)	N=4(bits)
Boosting bits in memory request queue entries	32 x N	128
Counters for number of requests per core in memory request queue	8 x N	32
Total storage required for P-NFQ	40 x N	160
P-PARBS		
Counters for number of requests per core in memory request queue	8 x N	32
Total storage required for P-PARBS	8 x N	32
P-FST		
Boosting bits in memory request queue entries	32 x N	128
Counters for number of requests per core in memory request queue	8 x N	32
Prefetch bits in pollution filter used for coordinated core and prefetcher throttling	Pol. Filter Entries (2048) x N	8192
Total storage required for P-FST	2088 x N	8352

Table 5.5: Hardware cost of our proposed enhancements

5.7 Conclusion

This chapter demonstrates a new problem in CMP designs: state-of-the-art fair shared resource management techniques, which significantly enhance performance/fairness in the absence of prefetching, can largely degrade performance/fairness in the presence of prefetching. To solve this problem, we introduce general mechanisms to effectively handle prefetches in multiple types of resource management techniques.

We develop three major new ideas to enable prefetch-aware shared resource management. We introduce the idea of *demand boosting*, a mechanism that eliminates starvation of applications that are not prefetch-friendly yet memory non-intensive, thereby boosting performance and fairness of any type of shared resource management. We describe how to intelligently prioritize demands and prefetches within the underlying fair management techniques. We develop new mechanisms to coordinate the actions of prefetcher and core throttling mechanisms to make synergistic decisions. To our knowledge, this is the first work that deals with prefetches in shared multi-core resource management, and enables such techniques to be effective and synergistic with prefetching.

We apply these new ideas to three state-of-the-art multi-core shared resource management techniques. Our extensive evaluations show that our proposal significantly improves system performance and fairness of two fair memory schedul-

ing techniques and the source-throttling-based shared memory system management technique we proposed in Chapter 4 (by more than 10% in 4-core systems), and makes these techniques effective with prefetching. We conclude that our proposal can be a low-cost and effective solution that enables the employment of both prefetching and shared resource management together in future multi-core systems. This will ensure future systems can reap the performance and fairness benefits of both ideas together.

Chapter 6

Parallel Application Memory Scheduling

6.1 Introduction

In Chapters 3 through 5 we presented mechanisms that address management of inter-application interference in the memory system for multi-programmed workloads. In addition to multi-programmed workloads, CMPs are also commonly used to speed up a single application using multiple threads that concurrently execute on different cores. Memory requests from concurrently executing threads can interfere with one another in the shared memory subsystem, slowing the threads down significantly. Most importantly, the *critical path* of execution can also be significantly slowed down, resulting in increased application execution time.

To illustrate the importance of DRAM-related inter-thread interference to parallel application performance, Figure 6.1 shows the potential performance improvement that can be obtained for six different parallel applications run on a 16-core system.[1] In this experiment, we ideally eliminate all *inter-thread* DRAM-related interference. Thread *i*'s DRAM-related interference cycles are those extra cycles that thread *i* has to wait on memory due to bank or row-buffer conflicts caused by concurrently executing threads (compared to if thread *i* were accessing the same memory system alone). In the ideal, unrealizable system we model for this experiment: 1) thread *i*'s memory requests wait for DRAM banks only if the banks are busy servicing requests from that same thread *i*, and 2) no DRAM row-conflicts occur as a result of some other thread *j* ($i \neq j$) closing a row that is accessed by thread *i* (i.e., we model each thread as having its own row buffer in each bank). This figure shows that there is significant potential performance to be obtained by

[1] Our system configuration and benchmark selection are discussed in Section 6.3.

better management of memory-related inter-thread interference in a parallel application: ideally eliminating inter-thread interference reduces the average execution time of these six applications by 45%.

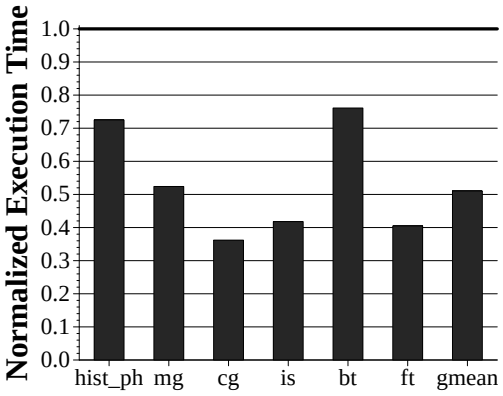


Figure 6.1: Normalized execution time

Chapters 3 through 5, and previous papers on managing memory system related *inter-application* interference [31, 36, 57, 32, 54, 51, 55, 17, 15, 37, 38, 16] address the problem of improving system performance (system throughput or average job turnaround time) and/or system fairness in the context of multi-programmed workloads where different cores of the CMP execute independent single-threaded applications.[2] None of these works directly address parallel *multi-threaded* applications as we do in this chapter where our goal of managing memory system inter-thread interference is very different: reducing the execution time of a single parallel application. Managing the interference between threads of a parallel application poses a different challenge than previous works: threads in a parallel application are likely to be inter-dependent on each other, whereas such inter-dependencies are assumed to be non-existent between applications in these previous works. Techniques for reducing inter-application memory interference for improving system performance and fairness of multi-programmed workloads may result in improved parallel application performance by reducing overall interference. However, as we

[2]We refer to interference between independent applications running on different cores as inter-application interference, and to interference between threads of a parallel application running on different cores as inter-thread interference.

show in this chapter, designing a technique that specifically aims to maximize parallel application performance by taking into account the inter-dependence of threads within an application can lead to significantly higher performance improvements.

Basic Idea: We design a memory scheduler that reduces parallel application execution time by managing inter-thread DRAM interference. Our solution consists of two key parts:

First, we propose estimating the set of threads likely to be on the critical path using *limiter thread* estimation (for lock-based synchronization) and *loop progress* measurement (for barrier-based synchronization). For lock-based synchronization, we extend the runtime system (e.g., runtime library that implements locks) with a mechanism to estimate a set of *limiter threads* which are likely critical (i.e., make up the critical path of the application). This estimate is based on lock contention, which we quantify as the time threads spend waiting to acquire a lock. For barrier-based synchronization used with parallel `for` loops, we add hardware iteration counters to estimate the progress of each thread towards the barrier at the end of the loop. We identify threads that fall behind as more likely to be critical.

Second, we design our memory controller based on two key principles: a) we prioritize threads that are likely to be on the critical path (which are either limiter threads or threads falling behind in parallel loops), and b) among a group of limiter threads, non-limiter threads, or parallel-for-loop threads that have made the same progress towards a synchronizing barrier (i.e. threads that are equally critical), we shuffle thread priorities in a way that reduces the time all threads collectively make progress.

6.2 Mechanism: Parallel Application Memory Scheduling

Our parallel application memory scheduler (PAMS):

1. Estimates likely-critical threads using limiter estimation (Section 6.2.1.1) and loop progress measurement (Section 6.2.1.2).

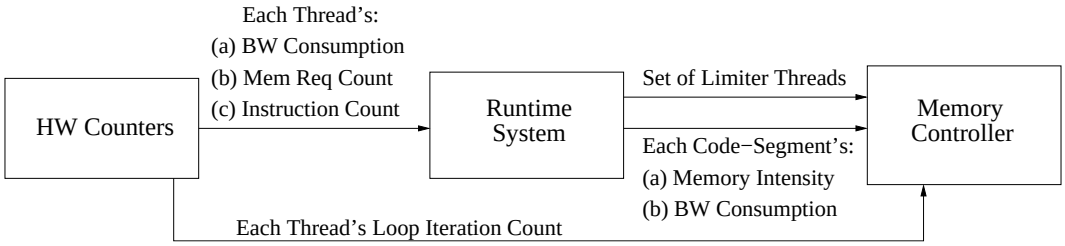


Figure 6.2: Overview of parallel application memory scheduling

2. Prioritizes likely-critical threads (Section 6.2.2.2) and shuffles priorities of non-likely-critical threads (Section 6.2.2.3) to reduce inter-thread memory interference.

Figure 6.2 provides an overview of the interactions between the major components of our design. The runtime system (e.g., runtime library that implements locks) uses hardware monitors to characterize memory behavior of *code-segments* (parts of the parallel program, see Section 6.2.2.1 for details) and passes this information to the memory controller. In addition, the runtime system provides the memory controller with a set of *limiter threads* (those likely to be on the critical path). Finally, the memory controller has access to iteration counts of parallel `for` loops. The following sections describe each component in detail.

6.2.1 Runtime System Extensions

In parallel applications, the *critical path* determines the execution time of the program. In each execution cycle, the critical path lies on one of the concurrently executing threads. Hence, to improve performance, the memory scheduler should minimize memory-related interference suffered by memory requests issued by the thread on the critical path. Unfortunately, identifying exactly which thread is on the critical path at runtime with low/acceptable overhead is difficult. However, we find that even a coarse estimation of the critical path can be very useful.

We propose to estimate the critical path via limiter thread estimation and loop progress measurement. Limiter thread estimation is a runtime system mechanism which identifies a set of threads likely to contain the thread on the critical

path by analyzing lock contention. We call these threads *limiter threads*, since one of them likely limits the application running time. Loop progress measurement is a cooperative compiler/hardware mechanism which estimates the progress of each thread within a parallel `for` loop, for programs structured with such barrier-synchronized loops across threads.

The memory controller uses limiter thread and loop progress information to manage inter-thread interference in the DRAM system and improve application performance.

6.2.1.1 Estimating Limiter Threads

When multiple threads concurrently execute and access shared data, correctness is guaranteed by the *mutual exclusion* principle: multiple threads are not allowed to access shared data concurrently. This mutual exclusion is achieved by encapsulating accesses to shared data in code regions guarded by synchronization primitives such as locks. Such guarded code is referred to as *critical section* code.

Prior work [68] shows that accelerating *critical sections* by executing them on high performance cores in a heterogeneous CMP can significantly reduce application running time. This is because contended critical sections are often on the *critical path*. We find that performance can be greatly improved by exposing information about contended critical sections to the memory controller, which uses this information to make better memory scheduling decisions. The rest of this subsection describes how this information is gathered by the runtime system and passed to the memory controller. We describe how the runtime system informs the memory controller of the single most contended critical section for ease of explanation; in general, however, the runtime system can detect any number of most contended critical sections.

As more and more threads contend over the lock protecting some shared data, it is more likely that threads executing the critical section guarded by the contended lock will be on the critical path of execution. As such, at a high level, the

runtime system periodically identifies the most contended lock. The thread holding that lock is estimated to be a limiter thread. Limiter thread information is passed to the memory controller hardware using the $LimiterThreadBitVector$ which has a bit per thread.[3] The runtime system identifies thread *i* as a *limiter* thread by setting the corresponding bit *i* in this bit-vector. This information is used by the memory controller in the following interval. The runtime system provides two main pieces of information which our algorithm uses to estimate limiter threads: the ID of the thread currently holding each lock, and the time a thread starts waiting for a lock.

Algorithm 9 explains limiter thread estimation in detail. The goal of the algorithm is to a) find the lock that causes the most contention in a given interval, and b) record the thread that owns this lock in $LimiterThreadBitVector$ so that the memory controller can prioritize that thread. To implement the algorithm, the runtime system maintains one counter per lock which accumulates the total cycles threads wait in that lock's queue, and keeps two variables to record the currently most-contended lock and the thread that owns it.

Every interval (i.e., every $LimiterEstimationInterval$ lock acquires), the runtime system finds the most-contended lock. To do so, it compares the lock queue waiting times accumulated for all of the locks. The system identifies the lock for which threads spent the most time waiting in the queue during the previous interval and saves it as $Lock_{longest}$. It then determines which thread is holding that lock, and sets the corresponding bit in the $LimiterThreadBitVector$.

To keep track of each lock's waiting time, every time a lock is successfully acquired by some thread i, the runtime system adds the time thread i spent waiting on the lock to the lock's waiting time counter (See Section 6.2.3 for implementation details). Finally, when a thread acquires the lock that had the longest waiting time in the previous interval ($Lock_{longest}$), $LimiterThreadBitVector$ is updated: the bit corresponding to the previous owner of the lock is reset in the vector, the bit for the

[3] In this chapter, we consider one thread of execution per core, but in systems with simultaneous multithreading (SMT) support, each thread context would have its own bit in $LimiterThreadBitVector$.

thread acquiring the lock is set, and the new owner is recorded as $LastOwner_{longest}$. This updated bit-vector is communicated to the memory controller in order to prioritize the limiter thread.

Algorithm 9 Runtime Limiter Thread Estimation

Every $LimiterEstimationInterval$ **lock acquires**
  Find lock with longest total waiting time in previous interval
  Set $Lock_{longest}$ to the lock with the longest waiting time
  Set $LastOwner_{longest}$ to the thread that holds $Lock_{longest}$
  Set bit for $LastOwner_{longest}$ in $LimiterThreadBitVector$
Every successful lock acquire
  Increment $waitingTime$ counter of acquired lock by the number of cycles spent in the lock's queue by the acquiring thread
  if acquired lock is $Lock_{longest}$ **then**
    Reset bit for $LastOwner_{longest}$ in $LimiterThreadBitVector$
    Record new $Lock_{longest}$ owner in $LastOwner_{longest}$
    Set bit for $LastOwner_{longest}$ in $LimiterThreadBitVector$
  end if

6.2.1.2 Measuring Loop Progress

Parallel `for` loops are a common parallel programming construct which allows for critical path estimation in a different way. Each iteration of a parallel `for` loop identifies an independent unit of work. These loops are usually statically scheduled by dividing iterations equally among threads. After the threads complete their assigned iterations, they typically synchronize on a barrier.

Given this common computation pattern, we can easily measure the progress of each thread towards the barrier by the number of loop iterations it has completed, as has also been proposed by Cai et al. [6]. We employ the compiler to identify this computation pattern and pass the address of the loop branch to the PAMS hardware. For each thread, we add a hardware loop iteration counter which tracks the number of times the loop branch is executed (i.e., the number of loop iterations completed by the thread) The runtime system resets these counters at every barrier.

The memory controller uses this loop progress information to prioritize

threads that have lower executed iteration counts, as described in Section 6.2.2.3.

6.2.2 Memory controller design

At a high level, our memory controller enforces three priorities in the following order (see Algorithm 10): First, we prioritize row-buffer hit requests over all other requests because of the significant latency benefit of DRAM row-buffer hits compared to row-buffer misses. Second, we prioritize limiter threads over non-limiter threads, because our runtime system mechanism deems limiter threads likely to be on the critical path. We describe prioritization among limiter threads in detail in Section 6.2.2.2. We prioritize remaining non-limiter threads according to *loop progress* information described in Section 6.2.1.2. Prioritization among non-limiter threads is described in detail in Section 6.2.2.3. Algorithm 10 serves as a high level description and outline for the subsections that follow.

Algorithm 10 Request Prioritization for
Parallel Application Memory Scheduler (PAMS)

1. Row-hit first
2. Limiter threads (Details of the following are explained in Section 6.2.2.2)
- Among limiter threads, latency-sensitive threads are prioritized over bandwidth-sensitive threads
- Among latency-sensitive group: lower-MPKI threads are ranked higher
- Among bandwidth-sensitive group: periodically shuffle thread ranks

3. Non-Limiter threads (Details of the following are explained in Section 6.2.2.3)
if loop progress towards a synchronizing barrier is known **then**
  - Prioritize threads with lower loop-iteration counts first
  - Among threads with same loop-iteration count: shuffle thread ranks
else
  - Periodically shuffle thread ranks of non-limiter threads
end if

6.2.2.1 Terminology

Throughout the subsections that follow, we will be using the term **code-segment** which we define as: a program region between two consecutive synchronization operations such as lock acquire, lock release, or barrier. Code-segments

starting at a lock acquire are also distinguished based on the address of the acquired lock. Hence, a code-segment can be identified with a 2-tuple:

<beginning IP, lock address (zero if code is not within a critical section)>

Code-segments are an important construct in classifying threads as latency- vs. bandwidth-sensitive (as we describe in the next subsection), and also in defining the intervals at which classification and shuffling are performed.

6.2.2.2 Prioritization among limiter threads

The goal for the limiter thread group is to achieve high performance in servicing the requests of the group, while also ensuring some level of fairness in progress between them as we do not know exactly which one is on the critical path. To this end, we propose classifying limiter threads into two groups: *latency-sensitive* and *bandwidth-sensitive*. Latency-sensitive threads (which are generally the less memory intensive threads) are prioritized over bandwidth-sensitive ones. As Algorithm 10 shows, among latency-sensitive threads, threads with lower MPKI are prioritized as they are less-memory intensive and servicing them quickly will allow for better utilization of the cores. Prioritization among bandwidth-sensitive threads is done using a technique called *rank shuffling* [38]. This technique is also used to prioritize non-limiter threads and, in fact, is more important in that context; hence, we defer discussion of rank shuffling to Section 6.2.2.3. The rest of this subsection describes how we classify threads as latency- vs. bandwidth-sensitive.

Latency-sensitive vs. bandwidth-sensitive classification of threads: As described in [38], a less memory intensive thread has greater potential to make progress and keep its core utilized than a more memory intensive one. Hence, classifying it as latency-sensitive and prioritizing it in the memory controller improves overall system throughput because it allows the thread to quickly return to its compute phase and utilize its core. To do this classification, the main question is how to predict the future memory intensity of the code a thread is about to execute.

We propose classifying threads as latency- or bandwidth-sensitive based on

the memory intensity of the *code-segment* that thread is executing. The key idea is that we can estimate the memory intensity of the code-segment that the thread is entering based on the memory intensity of that code-segment last time it was executed. Figure 6.3 illustrates this strategy. Classification of threads is performed at each code-segment change (indicated by a vertical dotted line in the figure). Algorithm 11 presents the details of the classification algorithm used by the memory controller. This algorithm is a modified version of the original thread clustering algorithm by Kim et al. [38] adapted to be invoked at every code-segment change.[4] The algorithm requires information about the memory intensity (number of misses per thousand instructions) and bandwidth consumption of the code-segment to be executed (number of cycles that at least one memory bank is busy servicing the code-segment's requests).

Algorithm 11 sets aside a fraction (*ClusterThreshold*) of the total bandwidth per cycle for latency-sensitive threads. It uses previous bandwidth consumption of currently executing code-segments to predict their current behavior. To do so, it sums up the previous bandwidth consumption of the least memory intensive currently-executing code-segments up to a *ClusterThreshold* fraction of total bandwidth consumption. The threads that are included in this sum are classified as latency-sensitive.

Note that in the original algorithm, Kim et al. [38] measure each cores' memory intensity every 10M cycles in a multi-core system where each core executes an independent application. In other words, they classify threads on a time interval basis rather than on the basis of a change in the code-segment. We find that with parallel workloads there is little information to be gained by looking back at a thread's memory behavior over a fixed time interval. Figure 6.4 shows why. In the figure, thread 2 spends a long time waiting on a lock in time quantum 2. However, its memory behavior measured during that time interval has nothing to do with its

[4]We refer the reader to Algorithm 1 in the original TCM [38] paper for details on the original algorithm.

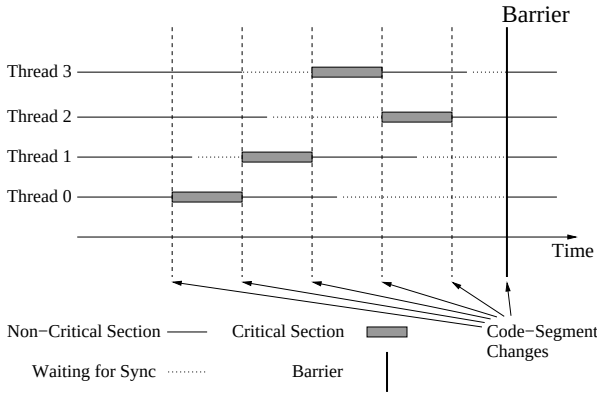


Figure 6.3: Code-segment based classification

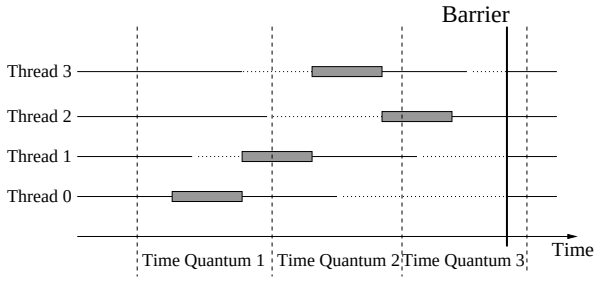


Figure 6.4: Time based classification

memory behavior in the following time interval (time quantum 3), during which it happens to be not waiting. For this reason, we perform classification on the basis of a code-segment change.

Keeping track of past code-segment memory behavior:

As a thread executes a given code segment, the memory controller maintains a counter for the number of memory requests generated by that code segment. Another counter maintains the number of instructions executed in the code segment. When the code segment ends, the runtime system takes control because a synchronization event has occurred. The runtime system reads both counters and calculates the memory intensity of that code segment which it stores for later use. It also keeps track of a *bandwidth consumed per cycle* count for the completed code segment. When that code segment is started on any thread in the future, the runtime system loads two registers in the memory controller with the memory intensity and

bandwidth consumed per cycle which were last observed for that code segment.

Algorithm 11 Latency-sensitive vs. Bandwidth-sensitive classification for limiter threads

Per-thread parameters:
$CodeSegMPKI_i$: MPKI of code-segment currently running on thread i the last time it occurred
$CodeSegBWConsumedPerCycle_i$: BW consumed per cycle by code-segment currently running on thread i the last time it occurred
$BWConsumed_i$: Bandwidth consumed by thread i during previous interval
Classification: (every code-segment change)
TotalBWConsumedPerCycle = ($\Sigma_i\ BWConsumed_i$) / $Length\ Of\ Previous\ Interval\ In\ Cycles$
while Threads left to be classified **do**
  Find thread with lowest MPKI (thread i)
  SumBW += $CodeSegBWConsumedPerCycle_i$
  if SumBW $\leq$ $ClusterThreshold$ $\times$ TotalBWConsumedPerCycle **then**
    thread i classified as $LatencySensitive$
  else
    thread i classified as $BandwidthSensitive$
  end if
end while

6.2.2.3 Prioritization among non-limiter threads

When the application is executing a parallel `for` loop, the memory controller uses loop progress information (Section 6.2.1.2) to ensure balanced thread execution. The measured loop progress information is used by the memory controller to create priorities for different threads in order of their loop progress: threads with lower iteration counts—those falling behind—are prioritized over those with higher iteration counts. This prioritization happens on an interval by interval basis, where the priorities assigned based on loop progress are maintained for a while to give threads that have fallen behind a chance to fully exploit their higher priority in the memory system (e.g., exploit row buffer locality). Subsequently, priorities are re-evaluated and assigned at the end of the interval for the next interval.

Among a set of threads that have the same loop progress or in the absence

of such information, the memory controller aims to service all bandwidth-sensitive threads in a manner such that none become a new bottleneck as a result of being deprioritized too much in the memory system. To achieve this, we perform interval-based *rank shuffling* of the threads.

Shuffling of bandwidth-sensitive threads:

At the beginning of each interval, we assign a random rank to each of the bandwidth-sensitive threads and prioritize their memory requests based on that ranking in that interval. The main question in shuffling the ranks of parallel threads is: when should an interval end and new rankings be assigned?

We find that a group of threads that have similar memory behavior should be treated differently than a group of threads that do not.[5] When threads have similar memory behavior, we find that maintaining a given random ranking until one of the threads finishes executing the code-segment it is currently executing can significantly improve performance. This is because when a code-segment ends (e.g., when the thread reaches a barrier), the inter-thread interference it was causing for the other threads is removed, and the other threads can make faster progress in its absence. We call this *code-segment based shuffling*: new thread ranks are assigned when a code-segment change happens. On the other hand, when a group of threads have very different memory behavior, we find that changing the thread ranking only on a code-segment change can sometimes lead to performance loss. For example, if the thread that is going to reach the barrier first is assigned the highest rank, keeping it prioritized until it reaches the barrier delays the thread that would be last to reach the barrier, lengthening the critical path of the program. As such, for threads with very different memory behavior, fixed-interval time-based shuffling of thread ranking performs better. This allows each thread to get quick service for its memory requests for a while and make proportional progress toward the barrier. We call this *time-based shuffling*.

[5]When the ratio between the largest memory intensity and the smallest memory intensity of all threads within a group of threads is small (less than 1.2 in our experiments), we refer to the group as a group of threads with similar memory behavior.

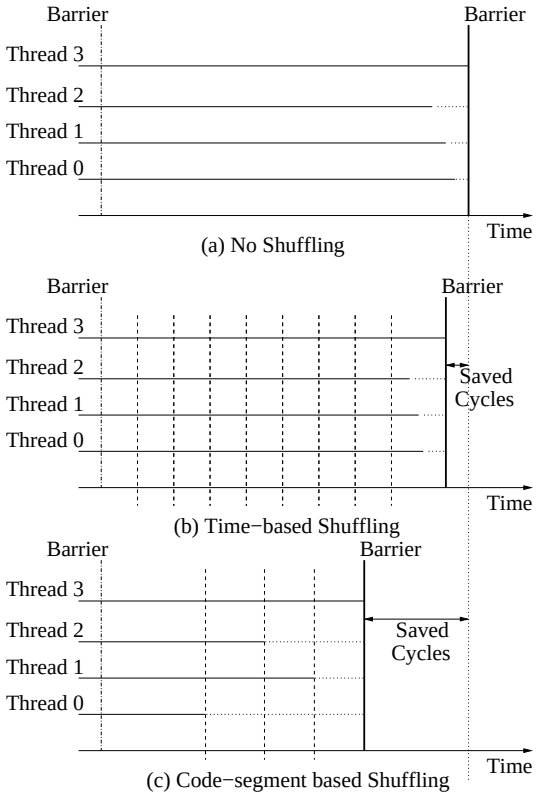


Figure 6.5: Threads have similar memory behavior

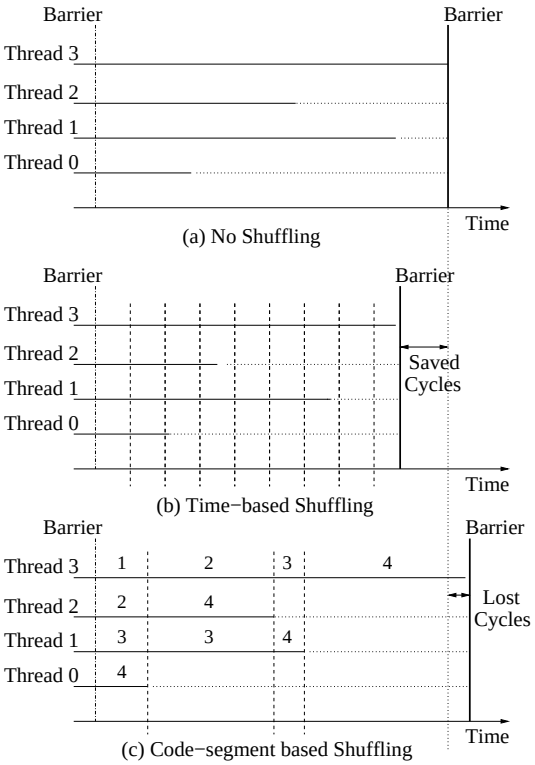


Figure 6.6: Threads have different memory behavior

Figures 6.5 and 6.6 illustrate how each of these two shuffling policies performs when applied to two very different scenarios for threads concurrently executing between two barriers.

When the set of threads have similar memory behavior as shown in Figure 6.5 (a), code-segment based shuffling can be significantly better than time-based shuffling. Behavior similar to this exists in the applications *ft* and *is*. Time-based shuffling (Figure 6.5 (b)) improves performance over no shuffling by allowing different threads to be prioritized during different time intervals and thus make proportional progress toward the barrier. However, all threads continue to interfere with one another in the memory system until they all reach the barrier at a similar time. Code-segment based shuffling reduces this interference between threads by ensuring some threads reach the barrier earlier and once they reach the barrier, they stop

exerting pressure on the memory system. As shown in Figure 6.5 (c) and described above, maintaining a given random ranking until a code-segment change happens (i.e., a thread reaches a barrier) allows the prioritized thread to reach its barrier before the deprioritized one. After that, the deprioritized thread can make much faster progress because previously-prioritized threads stop exerting memory interference as they are waiting at the barrier. For this very reason, code-segment based shuffling can significantly improve performance over time-based shuffling, as shown in the longer "Saved Cycles" of Figure 6.5 (c) compared to that of Figure 6.5 (b).

When the set of threads have different memory behavior as shown in Figure 6.6 (a), time-based shuffling can outperform code-segment based shuffling. Behavior similar to this can be observed in the *mg* application. With time-based shuffling (Figure 6.6 (b)), threads are assigned different random rankings for each fixed-length interval, which allows each thread to get quick service for its memory requests for a while. This reduces the time it takes for all threads to get to the barrier at the end of the interval. Figure 6.6(c) shows how code-segment based shuffling can easily perform poorly. The numbers shown above the threads in the different intervals are an example of random ranks assigned to the threads every time one of the threads' code-segment finishes (i.e., every time a thread reaches the barrier, in this example). Because the threads which would have reached the barrier earlier end up receiving a higher rank than the thread that would reach the barrier last (thread 3) after every code-segment change, code-segment based shuffling delays the "critical thread" by causing more interference to it. This results in performance loss compared to time-based shuffling and even compared to no shuffling, as shown in "Lost Cycles" in Figure 6.6(c).

Dynamic Shuffling Policy: Since neither of the two policies always performs best, we propose a dynamic shuffling policy that chooses either time-based shuffling or code-segment based shuffling based on the similarity in the memory behavior of threads. Our dynamic shuffling policy operates on an interval-basis. An interval ends when each thread executes a threshold number of instructions (we empirically determined this interval as 5000 instructions). Our proposed policy continuously

monitors the memory intensity of the threads to be shuffled. At the end of each interval, depending on the similarity in memory intensity of the threads involved, the memory controller chooses a time-based or code-segment-based shuffling policy for the following interval. As we will show in Section 6.4, this policy performs better than either time-based shuffling or code-segment based shuffling employed for the length of the application.

6.2.3 Implementation Details

Table 6.1 breaks down the modest storage required for our mechanisms, 1552 bits in a 16-core configuration. Additionally, the structures we add or modify require little energy to access and are not accessed very often. As such, significant power overhead is not introduced.

PAMS	Closed form for N cores (bits)	N=16 (bits)
Loop iteration counters	32 x N	512
Bandwidth consumption counters	16 x N	256
Number of generated memory requests counters	16 x N	256
Past code-segment information registers	2 x 16 x N	512
Limiter thread bit-vector	N	16
Total storage required for PAMS	97 x N	1552

Table 6.1: Hardware storage cost of PAMS

Limiter Estimation: In Algorithm 9, to keep track of the total time all threads spend waiting on lock l in an interval, we modify the runtime system (i.e., the threading library) to perform the following: When any thread attempts to acquire lock l, a timestamp of this event is recorded locally. Once lock l is successfully acquired by some thread i, the runtime system adds the waiting time for that thread (obtained by subtracting the recorded timestamp for thread i from the current time) to the waiting time counter of lock l. Note that the waiting time counter for lock l is protected by the lock itself as it is only modified by a thread once that thread has successfully acquired the lock.

The overhead of the runtime limiter estimation described in Algorithm 9 is insignificant as it does not occur very often. In our evaluations we empirically determine $LimiterEstimationInterval$ to be equal to five. Among our benchmarks, *hist* has the highest frequency of lock acquires, averaging one lock acquire every 37k cycles. Assuming sixteen locks are being tracked, the limiter estimation algorithm incurs the latency of sorting sixteen waiting times (each a 32-bit value) once every 185k cycles. A back-of-the-envelope calculation shows that this latency adds an overhead of less than 1% (even for the benchmark that has the most frequent lock acquires).

Alternative Hardware-Based Limiter Estimation: Even though the overhead of tracking total waiting time for each lock in the runtime system is very small in our implementation and evaluation, it could become more significant in the context of a locking library that is highly-optimized for fine-grain synchronization and when there is high lock contention. An alternative implementation of our proposal could track waiting time in hardware to further reduce the overhead. Although we did not evaluate this alternative, we outline its general idea here. In this implementation, two new instructions delimit the beginning and the end of each thread's wait for a lock: $LOCK\_WAIT\_START$ <*lock_address*> and $LOCK\_WAIT\_END$ <*lock_address*>. Each instruction takes a lock address, and updates a centralized lock table after commit, i.e. off the critical path.

This table contains one entry for each lock which contains the current number of threads waiting on that lock (*num_wait*) and the associated cumulative waiting time (*wait_time*). $LOCK\_WAIT\_START$ increments *num_wait* and $LOCK\_WAIT\_END$ decrements *num_wait* for the specified lock. Periodically, the hardware increments *wait_time* by *num_wait*, and estimates the limiter by finding the lock with the the highest *wait_time* and storing its address in a $Lock_{longest}$ register associated with the lock table. Since $LOCK\_WAIT\_END$ executes right before a thread starts the critical section, the instruction also compares the lock address with $Lock_{longest}$ and in case of a match, it reports the thread ID to the memory

controller as the current owner of $Lock_{longest}$, and the memory controller prioritizes requests from this thread.

6.3 Methodology

6.3.1 Processor Model

Table 6.2 shows the baseline configuration of each core and the shared resource configuration for the 16-core CMP system we use in the evaluations of this chapter. We faithfully model cache coherence, port contention, queuing effects, bank conflicts, and other major memory system constraints.

Execution core	15 stage out of order processor, decode/retire up to 2 instructions Issue/execute up to 4 micro instructions; 64-entry reorder buffer
Front end	Fetch up to 2 branches; 4K-entry BTB; 64K-entry Hybrid branch predictor
On-chip caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line ; L1 D-cache: 32KB, 4-way, 2-cycle, 64B line Shared unified L2: 4MB , 16-way, 16-bank, 20-cycle, 1 port, 64B line
DRAM controller	On-chip, FR-FCFS [65] scheduling 128-entry MSHR and memory request queue
DRAM and bus	667MHz bus cycle, DDR3 1333MHz [50] 8B-wide data bus, 8 DRAM banks, 16KB row buffer per bank Latency: 15-15-15ns; 100-100-100 processor cycles (^tRP-^tRCD-CL), Round-trip L2 miss latency: Row-buffer hit: 36ns, conflict: 51ns

Table 6.2: Baseline system configuration

6.3.2 Benchmarks

We use a selection of benchmarks from NAS Parallel Benchmarks (NPB 2.3) [12] and the *hist* benchmark from Phoenix [64]. For each NPB benchmark, we manually choose a representative execution interval delimited by global barriers (Table 6.3 lists the barriers used). We do this in order to simulate a tractable number of instructions with a large enough input set that will produce a meaningful number of memory requests. However, this was not possible for three of the NAS benchmarks *ep*, *lu*, and *sp*. This is because, with a large enough input set, we were unable to pick a tractable execution interval. We run the *hist* benchmark to

completion.

All benchmarks are compiled using the Intel C Compiler with the `-O3` option. Table 6.3 summarizes the benchmarks. The memory intensity values reported in this table are obtained from simulations on the system described by Table 6.2. The benchmarks we evaluate use Pthreads and OpenMP threading libraries. We modify the threading library to intercept library calls and detect locks. Also, we assume gang scheduling [60, 21] of threads where all the threads of a parallel application are concurrently scheduled to execute. As a result, thread preemption does not skew the threads' measured waiting times.

Benchmark	**Description**	**Input Set**	**Length**	**MPKI**	**Critical Sections**	**Barriers**	**Barrier Interval**
hist	Histogram (Phoenix)	minis	50M	2.66	405	1	N/A
mg	Multigrid solver (NPB)	W	225M	4.07	0	300	201–501
cg	Conjugate gradient solver (NPB)	A	113M	22.26	256	60	31–91
is	Integer sort (NPB)	W	140M	17.32	112	25	1–26
bt	Block tridiagonal solver (NPB)	W	397M	6.45	0	310	171–481
ft	Fast fourier transform (NPB)	W	161M	5.41	16	5	21–26

Table 6.3: Benchmark summary

6.3.3 Parameters Used in Evaluations

Table 6.4 shows the parameter values we use in our evaluations.

$LimiterEstimation\ Interval$	TCM Time Quanta	TCM Shuffling Period	Time-based Period (also used within Dynamic Shuffling)	Instruction Sampling Period in Dynamic Shuffling
5	2M cycles	100k cycles	100k cycles	5k insts

Table 6.4: Parameters used in evaluation

6.4 Results and Analysis

We first present performance results for each of the 6 benchmarks on a 16-core system normalized to their performance on a system using an FR-FCFS memory scheduler. Figure 6.7 shows results for the following six configurations from left to right for each benchmark, with each succeeding configuration introducing only one new component to the previous configuration: 1) thread cluster memory scheduling (TCM) [38], which uses time-based classification of latency-sensitive vs. bandwidth-sensitive threads with time-based shuffling, 2) code-segment based classification of latency-sensitive vs. bandwidth-sensitive threads (Section 6.2.2.2) with time-based shuffling, 3) code-segment based classification of threads with code-segment based shuffling (Section 6.2.2.3), 4) limiter information based thread prioritization (Section 6.2.1.1) with code-segment based classification and code-segment based shuffling, 5) limiter information based thread prioritization with code-segment based classification and dynamic shuffling policy, and 6) the combination of all our proposed mechanisms (PAMS): limiter information based thread prioritization, code-segment based thread classification with dynamic shuffling policy, and loop progress measurement based thread prioritization (note that no configuration except for this last one takes into account loop progress information in barrier based synchronization, described in Section 6.2.1.2). We find that among all evaluated mechanisms, PAMS provides the best performance, reducing execution time by 16.7% compared to a system with FR-FCFS memory scheduling, and by 12.6% compared to TCM, a state-of-the-art memory scheduling technique. Several observations are in order:

1. Applying TCM, which is a memory scheduling technique primarily designed for improving system performance and fairness in multi-programmed workloads, to parallel applications improves average performance by 4.6%. This is because even though this technique does not consider inter-dependencies between threads, it still reduces inter-thread memory system interference, providing quicker service to threads (average memory latency reduces by 4.8%), thus enabling faster application progress.

2. Using code-segment based classification of latency-sensitive vs. bandwidth-sensitive threads (second bar from the left for each benchmark) as explained in Section 6.2.2.2 improves performance significantly compared to the time-based classification done by TCM on two of the shown benchmarks (*hist* and *ft*). This is mainly because by using code-segments as interval delimiters to classify threads as latency- vs. bandwidth-sensitive (See Figure 6.3), we can make a more accurate classification of the thread's future memory behavior using information from the last time the starting code-segment executed.

3. When code-segment based shuffling is used instead of time-based shuffling (third bar from left, compared to second), performance improves significantly on three benchmarks (*hist*, *is*, and *ft*). This is primarily due to behavior shown in Figure 6.5. As explained in Section 6.2.2.3, when the group of concurrently executing threads have similar memory behavior, using code-segment based intervals for shuffling thread rankings outperforms time-based shuffling. On the other hand, in benchmarks *mg* and *cg*, execution time increases by as much as 6.8% (for *mg*) when code-segment based shuffling is used. This is because the threads have significantly different memory behavior, which can lead to performance degradation with code-segment based shuffling, as shown in Figure 6.6 (c). However, because of large improvements on *hist* (11%), *is* (14%), and *ft* (10%), average performance with code-segment based shuffling improves by 3.9% compared to time-based shuffling.

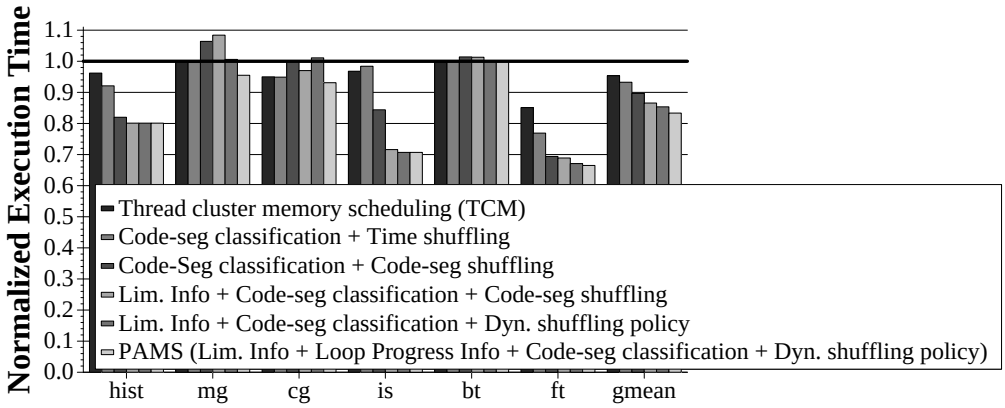


Figure 6.7: Overall Results

4. When limiter information is used to prioritize threads likely to be on the critical path (fourth bar from left), as described in Section 6.2.1.1, further benefits can be gained on applications that have contended locks. This can be seen in benchmarks such as *hist* and *is*. In these applications (one of which we will analyze in detail in a case study in Section 6.4.1), memory requests from limiter threads estimated by the runtime system are prioritized over non-limiter threads' requests, resulting in further execution time reduction. Note that when limiter information is used (in the three rightmost bars of Figure 6.7), latency- vs. bandwidth-sensitive classification of threads is performed only for limiter threads (as described by Algorithm 11 in Section 6.2.2.2).

5. Using the dynamic shuffling policy described in Section 6.2.2.3 (fifth bar for each benchmark) mitigates the performance loss seen due to code-segment based shuffling on benchmarks that have threads with different memory behavior, such as *mg* and *cg*. The dynamic shuffling policy monitors the memory intensity of concurrently executing threads and dynamically chooses code-segment based shuffling (when threads have similar intensities) or time-based shuffling (when threads have different intensities). With our dynamic shuffling policy, time-based shuffling is used for 74% and 52% of the time on *mg* and *cg* respectively.

6. *mg* and *cg* are also the benchmarks that benefit the most from prioritization of lagging threads enabled by loop progress measurement. This is expected since parallel `for` loops dominate the execution time of both benchmarks. In fact, *mg* and *cg* have very few critical sections, leaving loop progress measurement as the only way to estimate the critical path. Hence, performance of both benchmarks improves the most when loop progress measurement is enabled (4.5% and 6.9% over FR-FCFS, respectively).

6.4.1 Case Study

To provide insight into the dynamics of our mechanisms, we use the benchmark *is*, which has a combination of barriers and critical sections, as a case study.

This benchmark performs a bucket sort, each iteration of which consists of two phases: counting the integers belonging to each bucket and partially computing the starting index of each integer in the sorted integer array. The first phase is done in parallel; the second, however, modifies a shared array of partial results and hence requires a critical section. Figures 6.8(a)–(d) show *thread activity* plots generated by running *is* on the following configurations: a baseline system with an FR-FCFS memory controller, a system with TCM [38], a system that uses code-segment based shuffling and code-segment based classification of latency-sensitive vs. bandwidth-sensitive threads, and finally a system using our proposed PAMS.

In each *thread activity* plot shown in Figure 6.8, each thread's execution is split into three different states (as indicated by the legend on top of the figure): non-critical section execution (normal line), critical section execution (bold line), and waiting for a lock or barrier (dotted line). Vertical lines represent barriers where all threads synchronize.

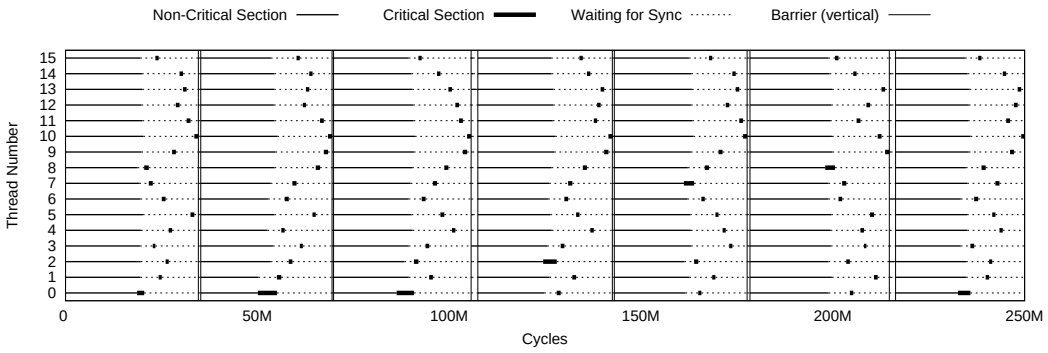
Several observations are in order: First, by using TCM [38], overall inter-thread interference is reduced compared to a baseline system with FR-FCFS, resulting in 3% reduction in execution time. This is mainly due to the reduction in execution time when threads are executing the non-critical section code that comes right after each barrier. This happens due to TCM's shuffling of priorities between the threads on time-based intervals, which leads to relatively similar improvement in the execution of all threads.

Second, performance can be significantly improved by using the code-segment based thread classification and shuffling that we propose in Sections 6.2.2.2 and 6.2.2.3 respectively. Figure 6.8c is a good real benchmark example of the behavior shown in Figure 6.5. Comparing the intervals between each pair of barriers across Figures 6.8c and (b) clearly shows the benefits of code-segment based shuffling vs. time-based shuffling in a benchmark where parallel threads executing non-critical section code have similar memory behavior.

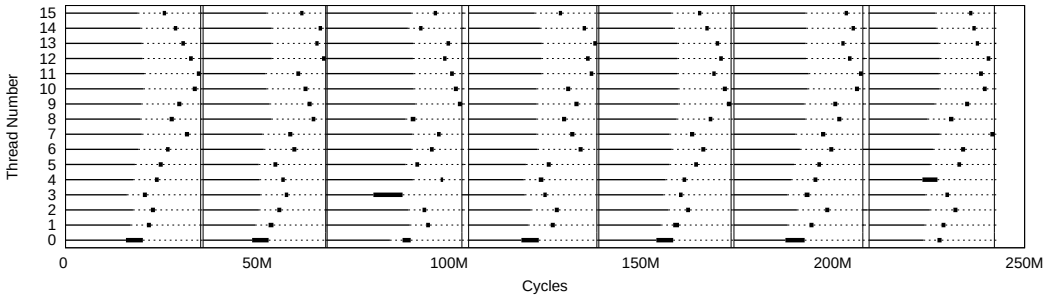
By keeping an assigned ranking constant until a code-segment change hap-

pens (which triggers the end of an interval and the assignment of a new ranking across threads) three benefits occur: 1) when a prioritized thread reaches the barrier, it starts waiting and stops interfering with other threads enabling their faster progress (as explained in Section 6.2.2.3), 2) with time-based shuffling all threads reach the point where they attempt to acquire the lock at a similar time resulting in high contention and waiting for the lock. Code-segment based shuffling reduces this lock contention. As a result, accesses to the critical section are spread over time and the first thread to reach the lock acquire in each barrier interval gets to that point earlier than with time-based shuffling (as seen in Figure 6.8(c)), and 3) code-segment based shuffling enables some threads to reach the critical section earlier than others as opposed to all threads reaching it at the same time (the latter happens in Figures 6.8(a) and (b)). This leads to the overlapping of the critical section latency with the execution of non-critical section code, and ultimately a reduction in the critical path of execution. As a result of these three major benefits, using code-segment based shuffling reduces execution time by 15.6% and 12.8% compared to the FR-FCFS baseline and TCM respectively.

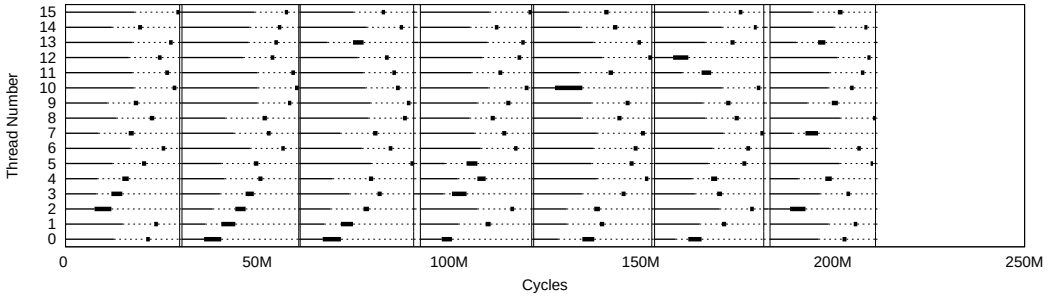
Finally, adding limiter information detected by the runtime system can significantly improve performance when combined with code-segment based classification and shuffling. Consider those critical sections that are part of the critical path in Figure 6.8c. As this figure shows, some threads enter their critical section early while other threads are still executing non-critical section code. Hence, memory requests from threads executing *non-critical* code can interfere with memory requests of the *critical* thread. However, by prioritizing memory requests from the thread identified as critical by the runtime system (Section 6.2.1), PAMS reduces the total time spent in the critical section by 29% compared to code-segment based classification and shuffling without limiter thread information (as shown by the improvement in Figure 6.8d compared to (c)). Overall, PAMS improves execution time by 28.4% and 26% compared to the FR-FCFS baseline and TCM respectively.



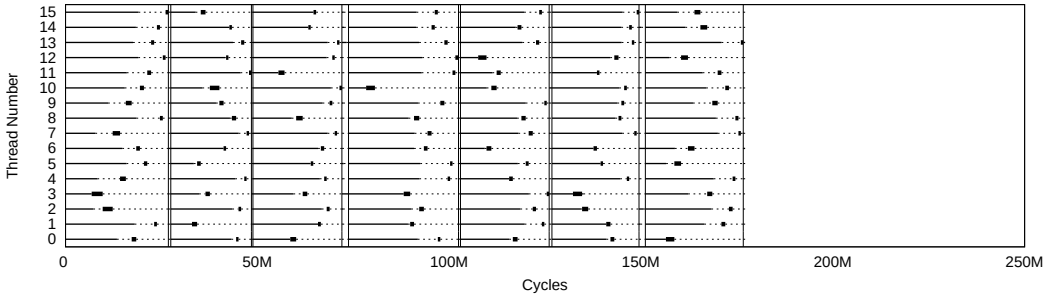
(a) FR-FCFS



(b) TCM



(c) Code-segment based classification and shuffling



(d) PAMS

Figure 6.8: Execution of *is* benchmark with different memory scheduling techniques

6.4.2 Comparison to Memory scheduling using Thread Criticality Predictors

Bhattacharjee and Martonosi [3] propose thread criticality predictors (TCP) to predict thread criticality based on memory hierarchy statistics. Although they do not demonstrate how their thread criticality predictor can be used for reducing inter-thread interference in the memory system, they do mention that it can be used in the design of memory controllers. We implement a memory scheduling technique based on the information TCP provides as a comparison point to PAMS. TCP uses L1 and L2 cache miss counts and the penalty incurred by such misses to determine a criticality count for a thread, defined in [3] as:

$$N(Crit.Count.) = N(L1miss) + \frac{LLCpenalty\ .\ N(LLCmiss)}{L1penalty}$$

In the TCP-based memory scheduling technique we developed, the criticality of each thread is obtained once every 100k cycles, and a set of rankings is assigned to threads based on their criticality. Threads with higher estimated criticality are given higher priority for that interval. At the end of each interval, thread criticalities are re-evaluated and a new set of priorities are assigned for the next interval. As Table 6.5 shows, we find that our technique, PAMS, outperforms this TCP-based memory scheduler by 6.6% on average. PAMS outperforms TCP significantly on three of the benchmarks. This improvement is mainly due to the following which TCP does not address: 1) PAMS uses information about the multi-threaded application such as lock contention and loop progress to estimate thread criticality, and 2) PAMS also addresses how to schedule requests of non-critical threads (e.g., shuffling of non-limiter bandwidth-sensitive threads). As such, the TCP idea is orthogonal to some of our proposals and could be used within PAMS as part of the basis for predicting critical/limiter threads, which we leave to future work.

Benchmark name	hist	mg	cg	is	bt	ft	gmean
Δ Execution time	-9.9%	-15.0%	-9.8%	-1.3%	0.2%	-2.5%	-6.6%

Table 6.5: Reduction in execution time of PAMS compared to TCP-based [3] memory scheduling

6.4.3 Sensitivity to System Parameters

Table 6.6 shows how PAMS performs compared to FR-FCFS and TCM on systems with 8MB/16MB shared last level caches or two/four independent memory channels. Even though using a larger cache or multiple memory channels reduces interference in main memory, PAMS still provides significantly higher performance than both previous schedulers. We conclude that our mechanism provides performance benefits even on more costly systems with higher memory bandwidth or larger caches.

Channels	LLC	Δ wrt FR-FCFS	Δ wrt TCM
Single	4MB	-16.7%	-12.6%
Single	8MB	-15.9%	-13.4%
Single	16MB	-10.5%	-5.0%
Dual	4MB	-11.6%	-10.0%
Quad	4MB	-10.4%	-8.9%

Table 6.6: Sensitivity of PAMS performance benefits to memory system parameters

6.5 Conclusion

We introduced the Parallel Application Memory Scheduler (PAMS), a new memory controller design that manages inter-thread memory interference in parallel applications to reduce overall execution time. To achieve this, PAMS employs a hardware/software cooperative approach that consists of two new components. First, the runtime system estimates likely-critical threads due to lock-based and barrier-based synchronization using different mechanisms and conveys this information to the memory scheduler. Second, the memory scheduler 1) prioritizes the likely-critical threads' requests since they are the performance bottleneck, 2) periodically shuffles the priorities of non-likely-critical threads to reduce memory interference between them and enable their fast progress. To our knowledge, PAMS is the first memory controller design that explicitly aims to reduce inter-thread interference between inter-dependent threads of a parallel application.

Our experimental evaluations show that PAMS significantly improves parallel application performance, outperforming the best previous memory scheduler

designed for multi-programmed workloads and a memory scheduler we devised that uses a previously-proposed thread criticality prediction mechanism to estimate and prioritize critical threads. We conclude that the principles used in the design of PAMS can be beneficial in designing memory controllers that enhance parallel application performance and hope our design inspires new approaches in managing inter-thread memory system interference in parallel applications.

Chapter 7

Conclusion and Future Research Directions

7.1 Conclusion

Inter-application memory system interference in multi-programmed workloads and inter-thread memory system interference in parallel multi-threaded workloads are major obstacles to high-performance and fair memory system design for CMPs. This dissertation identified significant shortcomings of conventional techniques for management of both *inter-application* and *inter-thread interference* in the shared memory subsystem. To overcome these shortcomings, we proposed and evaluated low-cost mechanisms for both types of interference. We proposed three mechanisms addressing different shortcomings of current designs in dealing with inter-application interference in multi-programmed workloads. We also proposed one mechanism which speeds up parallel multi-threaded workloads by managing DRAM-related interference between multiple threads of the same application.

To significantly improve the benefits of prefetching in CMP systems, this dissertation proposed hierarchical prefetcher aggressiveness control (HPAC). HPAC takes *prefetcher-caused inter-application interference* into account to determine the aggressiveness of each core's prefetcher. HPAC dynamically adjusts the aggressiveness of each prefetcher in two ways: *local* and *global*. The local decision attempts to maximize the local core's performance by taking into account only local feedback information. The global mechanism can override the local decision by taking into account effects and interactions of different cores' prefetchers when adjusting each one's aggressiveness. Chapter 3 shows that HPAC significantly improves system performance and bandwidth-efficiency compared to state-of-the-art prefetcher control techniques that do not take into account inter-application interference.

To provide fair sharing of the entire shared memory system to different applications without the complexity of developing fairness mechanisms for each individual resource, this dissertation proposes fairness via source throttling (FST). FST estimates unfairness in the entire shared memory system, and enforces system-software-defined fairness objectives by throttling cores accordingly via adjusting the number of requests they can inject into the system and the frequency at which they can do so. Chapter 4 shows that FST can significantly improve both system performance and fairness compared to state-of-the-art *resource-based* fairness techniques implemented independently for different shared resources.

This dissertation identified for the first time that, proposals which address high-performance and fair management of shared resources can significantly degrade both performance and fairness rather than improve them in the presence of prefetching. Chapter 5 proposed mechanisms that both manage the shared resources of a CMP to obtain high-performance and fairness, and also exploit prefetching. We apply these ideas to three state-of-the-art shared resource management techniques. Our evaluations show that these proposals significantly improve system performance and fairness of two fair memory scheduling techniques and our proposed FST technique from Chapter 4.

To reduce the execution time of parallel multi-threaded workloads, this dissertation proposes a memory controller design that takes into account information specific to parallel applications in designing the memory scheduling algorithm. Our parallel application memory scheduling (PAMS) mechanism from Chapter 6 consists of two components. First, estimating the critical path using *limiter thread* estimation and *loop progress* measurement. Second, a memory controller based on two principles: a) prioritizing threads likely to be on the critical path, and b) shuffling priorities among a group of limiter or non-limiter threads in a way that reduces the time it takes for them to reach a synchronization point. We show that this memory controller design significantly improves the performance of parallel applications compared to a state-of-the-art memory controller designed for multi-programmed workloads.

7.2 Future Research Directions

There are several possible future research directions that could improve the management of inter-application/thread interference for more fair and higher-performance memory system designs.

- The source-throttling-based management technique presented in Chapter 4 keeps the resource management techniques unchanged compared to the baseline in order to make the shared memory resource designs simpler. However, some performance-enhancing or fairness features of resource-based approaches could potentially be used in combination with FST's source-based approach to further improve performance and fairness.

- Our PAMS mechanism in Chapter 6 is targeted at DRAM-related inter-thread interference which is a major component of memory system inter-thread interference. However, management of interference among threads of a parallel application in other shared resources (e.g., shared caches, interconnect, etc.) could further improve parallel application performance. For instance the application of source-throttling-based shared resource management or a combination of resource-based and source-based techniques may provide further performance improvements.

- As industry continues to place more and more cores on the same chip (i.e., the emergence of many-core CMPs), CMPs will almost certainly be used to concurrently execute multiple multi-threaded applications which share parts of the memory system. To manage memory system interference in such systems, our solution for parallel multi-threaded applications (PAMS, Chapter 6) can be combined with existing solutions that deal with multiple applications (i.e., PAR-BS, TCM, or FST proposed in Chapter 4).

- With multiple concurrently executing multi-threaded applications on a many-core system, different system-software-specified fairness guarantees and performance goals can be of interest for the different applications. Combin-

ing software-based scheduling approaches [76, 69] (discussed in Chapter 2) with fine-grained source throttling (FST, Chapter 4) may be useful in satisfying different system-software goals by managing shared memory resources at different levels of granularity.

Bibliography

[1] Advanced Micro Devices. AMD's six-core Opteron processors. http://techreport.com/articles.x/17005, 2009.

[2] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.

[3] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA*, 2009.

[4] R. Bitirgen et al. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO-41*, 2008.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13, 1970.

[6] Q. Cai, J. Gonzalez, R. Rakvic, G. Magklis, P. Chaparro, and A. Gonzalez. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, 2008.

[7] F. J. Cazorla et al. QoS for high-performance SMT processors in embedded systems. *IEEE Micro*, 24(4):24–31, 2004.

[8] M. Charney and T. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 31(3):265–286, 1997.

[9] S. Chen et al. Scheduling threads for constructive cache sharing on CMPs. In *SPAA*, 2007.

[10] H.-Y. Cheng et al. Memory latency reduction via thread throttling. In *MICRO*, 2010.

[11] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X*, 2002.

[12] D. H. Bailey et al. NAS parallel benchmarks. Technical report, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, March 1994.

[13] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP-22*, 1993.

[14] R. Das et al. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.

[15] 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*, 2010.

[16] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt. Prefetch-aware shared resource management for multi-core systems. In *ISCA*, 2011.

[17] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO*, 2009.

[18] E. Ebrahimi, O. Mutlu, and Y. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, 2009.

[19] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[20] A. Fedorova et al. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, 2007.

[21] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *JPDC*, 16(4):306–318, 1992.

[22] R. Gabor et al. Fairness and throughput in switch on event multithreading. In *MICRO-39*, 2006.

[23] A. Gendler, A. Mendelson, and Y. Birk. A pab-based multi-prefetcher mechanism. *International Journal of Parallel Programming*, 34(2):171–478, Apr. 2006.

[24] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, Oct. 1998.

[25] D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Journal of Machine Learning*, 3(2-3):95–99, 1988.

[26] A. Herdrich et al. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS*, 2009.

[27] G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Feb. 2001. Q1 2001 Issue.

[28] L. R. Hsu et al. Communist, utilitarian and capitalist cache policies on cmps: caches as a shared resource. In *PACT*, 2006.

[29] Intel. First the tick, now the tock: Next generation Intel microarchitecure (Nehalem). *Intel Technical White Paper*, 2008.

[30] E. Ipek et al. Self-optimizing memory controllers: A reinforcement learning approach. In *MICRO*, 2008.

[31] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS*, 2004.

[32] R. Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.

[33] M. Jahre and L. Natvig. A light-weight fairness mechanism for chip multiprocessor memory systems. In *Computing Frontiers*, 2009.

[34] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA-24*, 1997.

[35] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990.

[36] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[37] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[38] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[39] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.

[40] H. Q. Le et al. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51:639–662, 2007.

[41] C. J. Lee et al. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.

[42] C. J. Lee et al. Improving memory bank-level parallelism in the presence of prefetching. 2009.

[43] C. J. Lee, O. Mutlu, V. Narasiman, and Y. Patt. Prefetch-aware DRAM controllers. In *MICRO-41*, 2008.

[44] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *ICPP-16*, 1987.

[45] J. Li et al. The thrifty barrier: energy-aware synchronization in shared memory multiprocessors. 2004.

[46] W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In *ICCD*, 2001.

[47] Y.-J. Lin et al. Hierarchical memory scheduling for multimedia mpsocs. In *ICCAD*, 2010.

[48] C. Liu et al. Exploiting barriers to optimize power consumption of CMPs. In *IPDPS*, 2005.

[49] K. Luo et al. Balancing throughput and fairness in SMT processors. In *ISPASS*, 2001.

[50] Micron. *Datasheet: 2Gb DDR3 SDRAM, MT41J512M4 - 64 Meg x 4 x 8 banks, http://download.micron.com/pdf/datasheets/dram/ddr3*.

[51] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.

[52] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-5*, 1992.

[53] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Using the first-level caches as filters to reduce the pollution caused by speculative memory references. *International Journal of Parallel Programming*, 33(5):529–559, October 2005.

[54] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[55] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.

[56] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *PACT*, 2004.

[57] K. J. Nesbit et al. Fair queuing memory systems. In *MICRO-39*, 2006.

[58] K. J. Nesbit et al. Virtual private caches. In *ISCA-34*, 2007.

[59] K. J. Nesbit et al. Virtual private machines: A resource abstraction for multi-core computer systems. Technical Report ECE 07-08, University of Wisconsin-Madison, Dec. 2007.

[60] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *IEEE Distributed Computer Systems*, 1982.

[61] J. Owen and M. Steinman. Northbridge architecture of AMD's Griffin microprocessor family. *IEEE Micro*, 28(2), 2008.

[62] H. Patil et al. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[63] 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*, 2006.

[64] C. Ranger et al. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[65] S. Rixner et al. Memory access scheduling. In *ISCA-27*, 2000.

[66] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *ASPLOS-IX*, 2000.

[67] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*, 2007.

[68] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

[69] L. Tang et al. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.

[70] J. Tendler et al. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[71] D. M. Tullsen and S. J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *ISCA-20*, 1993.

[72] O. Wechsler. Inside Intel core microarchitecure. *Intel Technical White Paper*, 2006.

[73] D. H. Woo and H.-H. S. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[74] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX*, 2009.

[75] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP-32*, 2003.

[76] S. Zhuravlev et al. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.

Vita

Eiman Ebrahimi was born in Tehran, Iran on September 16, 1982. He attended Kamal High School in Tehran, Iran until 2000. He started his Bachelors in Computer Engineering from the University of Tehran in 2000 and graduated in 2005. He moved to the United States to begin graduate school under the supervision of the late Professor Margarida Jacome in 2005. He received his Masters in 2007. He started working on his PhD with Professor Yale Patt in 2007.

While in graduate school, Eiman served as a teaching assistant for six semesters at The University of Texas at Austin: The mixed signal laboratory between 2005 and 2007, EE306 Introduction to Computing in Fall 2008, and EE382N Microarchitecture in Spring 2010. He completed three internships at the following companies: PA Semi, Microsoft Research, and IBM Research. He published papers in the International Symposium on Computer Architecture (ISCA), International Conference on Architectural Support for Programming Languages (ASPLOS), International Symposium on Microarchitecture (MICRO), International Symposium on High-Performance Computer Architecture (HPAC), and ACM Transactions on Computer Systems (TOCS). His honors include a best paper award at ASPLOS in 2010, and a best paper award nomination at HPCA in 2009.

Permanent address: 1440 First St., Kharazm St., Shahrak-e-Gharb,
Tehran, Iran

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.