The Pennsylvania State University
The Graduate School

REDUCING INTERFERENCE IN MEMORY HIERARCHY RESOURCES USING APPLICATION AWARE MANAGEMENT

A Dissertation in
Computer Science and Engineering
by
Sai Prashanth Muralidhara

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2011

The dissertation of Sai Prashanth Muralidhara was reviewed and approved* by the following:

Mahmut Taylan Kandemir
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Padma Raghavan
Professor of Computer Science and Engineering

Mary Jane Irwin
Professor of Computer Science and Engineering

Qian Wang
Associate Professor of Mechanical and Nuclear Engineering

Raj Acharya
Head of Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Aggressive technology scaling has resulted in an increase in number of cores being integrated on-chip. While on-chip cores are increasing at a fast rate, the memory hierarchy resources are scaling at a much slower pace. The memory resources, such as different levels of on-chip cache and the off-chip memory bandwidth, are costly and are often shared across multiple on-chip cores. This leads to multiple applications contending for access to these common resources. In the process, these applications can harmfully interfere with one another, and, this interference can result in significant degradation of both system throughput and individual application performance. Therefore, intelligently managing the shared memory resources by mitigating *inter-application interference* is vitally important in emerging multicore systems.

This dissertation makes three key contributions towards addressing the above problem of interference in shared memory resources. First, this dissertation considers the last-level shared cache and the off-chip memory as instances of shared memory resources, and, studies the causes and different ways in which applications interfere with one another while contending for a resource. Second, this dissertation studies the negative impact of resource contention on application and system performances. Third, this dissertation proposes novel schemes to mitigate inter-application interference and thereby improve system and application performance. These schemes aim to efficiently manage the resources in an *application aware* manner with the goal of mitigating the overall inter-application interference. An application aware resource management scheme considers the memory access characteristics of all the contending applications and uses this information to manage the shared resource. The resource management decisions are based on two key principles: 1) isolating applications/threads that harmfully interfere from each other by partitioning the resources between the interfering applications, and, 2) deciding the size of the resource partition that an application gets based on its memory access characteristics and requirements.

The trend of integrating increasing number of cores on a single chip is projected to continue into the future. This continued scaling is propelling the parallel computation capability of emerging multicore systems. Efficient management of shared memory hierarchy resources will become ever more important in the future if we are to ensure that applications extract the maximum possible parallelism from these multicore systems. This dissertation takes an important step towards addressing this problem by proposing novel schemes to efficiently manage multiple memory hierarchy resources. These schemes are very effective in practice, improving both system performance and individual application performance.

Table of Contents

Chapter 5
Intra-Application Cache Partitioning 68

Chapter 6
Bandwidth Constrained Coordinated HW/SW Prefetching for Multicores 83

List of Figures

List of Tables

Acknowledgments

I sincerely thank my advisor, Professor Mahmut Kandemir. He has advised and supported me immensely during my entire PhD stint. I learnt valuable lessons in research and technical writing from him. He patiently reviewed my papers multiple times and always provided valuable comments. He also provided me ample freedom to pursue research projects that interested me. For this, I am extremely thankful to him.

I am also thankful to my dissertation committee members, Professor Padma Raghavan, Professor Mary Jane Irwin, and Professor Qian Wang, for their valuable time and comments. I am especially thankful to professors, Padma Raghavan and Mary Jane Irwin. Interactions with Dr. Raghavan and her research group have always been interesting and lively. Discussions with Dr. Irwin have always been insightful and immensely helpful at various times during my research.

I am also extremely appreciative and thankful to my fellow MDL lab mates, and other colleagues in the department for very fruitful technical discussions, lively debates and friendly chats. During this process, I am glad that I made some good friends.

I would also like to thank Microsoft Research and Intel Corporation for providing me summer internship opportunities that proved invaluable. Being mentored by Bruce Christenson and Kathakali Debnath at Intel was a great experience.

I owe my deepest gratitude to my parents. My parents, Dr. Madanapalli Krishnamurthy Muralidhara and Geetha Kandi, have been an inspiration throughout my life. They are both dedicated teachers. They taught me the value of education by setting an example through their own careers and lives. But for their unflinching love, support and inspiration, this PhD would not have been possible. I am deeply indebted to them. I have valued their and my sister, Prathima Muralidhara's support immensely throughout my graduate career. I believe that my identity is due in part to my family lineage and heritage. I express my sincere gratitude to my late grandparents, who achieved so much during more difficult and challenging times. They imparted some memorable nuggets of wisdom that have stayed with me for

life. I am also thankful to my in-laws for their constant support.

My wife, Rashmi Murthy, has been right beside me through all the ups and downs in the last five years. It is her love and support that got me through difficult times. Throughout my haphazard schedules and paper deadlines, I always leaned on her for support and understanding. For her love and belief in me, I cannot be more thankful.

Dedication

To my family

Chapter 1

Introduction

The hardware industry is continuing on the path of chip multiprocessing [2, 3, 4]. This trend of increasing number of on-chip cores is projected to continue in the future. The current Intel Xeon processors [2] with six to eight cores on a chip, and Tilera's hundred core chip multiprocessor (CMP) are good indicators of the future of this trend. Increasing core counts translate to increasing parallel computation capability of current and future multicore processors.

While this scaling in core counts continues, the memory hierarchy resources are scaling at a much slower rate. As a result, the memory resources remain costly and limited. Two key memory hierarchy resources are the *last level cache*, and, the *off-chip memory*. The last level cache is the last line of defense before a memory request goes off-chip, making it a critically important resource. In the case of off-chip memory, the number of pins available for off-chip communication is limited by chip dimensions. More critically, the typical off-chip memory access latencies are one to two orders of magnitude higher than on-chip cache access latencies.

In order to improve efficiency of utilization, both of these resources are generally shared across all or a subset of on-chip cores in current multicore systems. Sharing enables multiple cores to use these costly resources flexibly and improve net resource utilization. In these systems, different levels of caches are shared to different degrees by the cores forming an on-chip cache hierarchy [5, 6]. Off-chip memory, on the other hand, is generally shared across all the on-chip cores [7, 8].

1.1 Problem: Interference in Memory Hierarchy Resources

Due to being shared, different applications/threads executing on different cores contend for access to the common memory resources. In the process, the contending applications/threads interfere with one another. This interference, termed "*inter-application interference*", can negatively impact the performance of all or some of the contending applications. In order to improve system performance and allow all contending applications to make fast progress, the memory resource needs to be managed such that this harmful inter-application interference is mitigated. The last level cache and off-chip memory play a key role in determining performance, and consequently, managing them efficiently to mitigate the negative impact of interference is vitally important.

1.1.1 Shared Cache Interference

When two cores share a cache, an application executing on one of the cores can evict a cache line belonging to the other application at any point during the execution. This *inter-core cache interference* can lead to the performance degradation of either or both applications if the applications continue to evict each other's data during the execution.

In order to study the impact of cache interference on performance, we executed a SPEC 2006 application, *bzip*2, with a companion application on a system with the cores sharing a last level cache. The applications, *bzip*2 and the companion application were run on two different cores, and this experiment was repeated with *bzip*2 running with different companion applications each time. For this evaluation, the off-chip memory was modeled with a constant latency access in order to isolate the cache interference effects.

Figure 1.1 shows the results of this series of experiments. The performance of *bzip*2 is affected to different degrees when executed with different companion applications. We observed that, this was due to different companion applications interfering with *bzip*2 to different degrees. This in turn is due to the memory access characteristics of the companion applications, as well as that of *bzip*2. With the

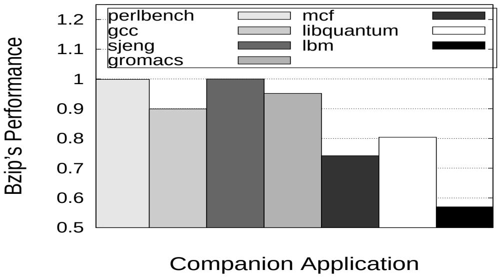


Figure 1.1. *Bzip*'s performance when executed with different companion applications. *Bzip* and the companion application are run on a two-core system with the cores sharing a last level cache. Performance results are normalized to the highest performance case.

knowledge of the way applications interfere, system performance can be improved by isolating applications into groups and partitioning the resources among these groups. For instance, from Figure 1.1, it is easy to determine that, *bzip*2 needs to be isolated from *mcf*, *lbm*, and *libquantum*, in order to improve *bzip*2's performance. On the other hand, *bzip*2 can be grouped together with *perlbench* and *gromacs* with no or negligible loss in Bzip's performance.

The effects of cache interference can be more complex when interfering threads belong to the same application. To start with, threads from the same application can share data. Consequently, an application can fetch data into the shared cache and this data can potentially be used by the other thread later in the execution. In this case, shared cache leads to *constructive cache sharing.* Therefore, in the multithreaded application case, there is a tradeoff between negative effects of inter-thread interference and the positive impact of cache sharing. This trade-off is considered and evaluated in Chapter 5.

1.1.2 Off-Chip Memory Interference

When a core's memory request misses in the last-level cache, that request enters the memory controller queue in order to be granted access to the off-chip memory. When two applications executing simultaneously generate memory requests, the memory requests from the two applications can interfere in memory controller queue, increasing the queuing time of either or both application's requests. The net performance degradation of applications in this scenario depends on the rate at which applications generate memory requests and other memory access characteristics of the applications such as row buffer locality (see Section 3.3 in Chapter 3).

In order to study the effect of off-chip memory interference on performance, we ran four copies each of *bzip*2 and a companion application on an eight core system with two memory channels. In this evaluation, we consider private last level caches in order to isolate memory interference effects.

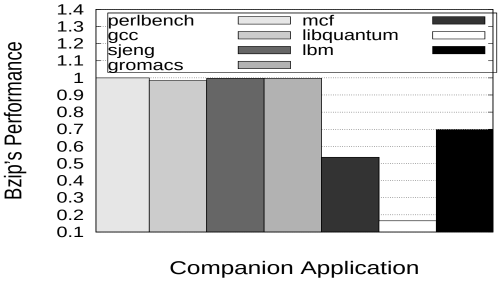


Figure 1.2. *Bzip*'s performance when executed with different companion applications. Four copies each of *Bzip* and the companion application are run on an eight-core system with the cores sharing two off-chip memory channels. Performance results are normalized to the highest performance case.

Figure 1.2 shows *bzip*2's average performance when the companion application is varied. From Figure 1.2, the performance of *bzip*2 suffers greatly due to interference from *mcf*, *lbm*, and *libquantum*. As in the shared cache case, other companion applications do not destructively interfere with *bzip*2 in any significant manner.

We discuss memory interference and its performance effects in more detail in Chapter 3.

1.2 Approach: Application Aware Management

This dissertation addresses the problem of efficiently managing a shared memory hierarchy resource. We first study, in detail, the application interference effects at the last-level shared cache and the off-chip memory. We then evaluate the impact of this interference on application and system performances. This dissertation then proposes novel schemes to efficiently manage the last-level cache and the off-chip memory bandwidth. These schemes mitigate the interference effects by managing the memory resource in an *application aware* manner. By application aware, we mean that the resource management schemes are aware of the memory access characteristics of all contending applications. In order to motivate the need for application aware management, we present the following three key insights:

1. From Sections 1.1.1 and 1.1.2, we can see that, application performance can be severely affected by inter-application interference. Therefore, mitigating interference needs to be the primary concern in managing a shared resource. Isolating an application from other applications during execution can be beneficial in improving its performance. However, this is not always true. Some applications can be grouped with another companion application with very little performance degradation. Therefore, isolation and grouping, when performed appropriately, can mitigate interference and improve performance.

2. Effect of inter-application interference on performance is not uniform and depends on which applications are contending for a resource. More specifically, we show later in this dissertation that, effect of interference on performance depends on specific characteristics of contending applications' memory access behavior. These characteristics include last level cache Misses Per Kilo Instruction (MPKI), reuse distance distribution, and, row buffer locality. Therefore, a shared resource management scheme needs to be *application aware* – aware of the memory access characteristics of all individual contending applications.

3. Memory demands of applications are different from one another. An application can be *latency sensitive*, which means that the application generates very few memory requests. If these rare memory requests are serviced quickly, the instruction window moves very fast since those few memory instructions are interspersed with a large number of compute instructions. On the other hand, an application can be *bandwidth sensitive*, which means that the application generates memory requests at a faster rate and therefore needs a constant bandwidth for the instruction window to move. While, both the effects of inter-application interference discussed previously, and, the memory demands of applications may depend on the same memory access characteristics of the applications, it is important to note the subtle difference between the two concepts.

In this dissertation, we propose multiple shared memory resource management schemes that are based on the three basic insights deduced above. As mentioned previously, our proposed mechanisms reduce inter-application interference by managing the memory hierarchy resource in an *application aware* manner. Specifically, the resource management schemes identify the memory access characteristics of individual applications, and, utilize this information to manage and allocate resources in order to mitigate interference. We show that, by managing the resources in this manner, both system and application performance is improved significantly.

1.3 Contributions

In this section, we introduce the specific contributions made by this dissertation in addressing the problem of shared memory interference.

In Chapter 3, we address the problem of mitigating interference at the off-chip memory. There are two general approaches to reduce this interference. One is to manage interference at the memory controller (e.g., memory scheduling), the other is to avoid interference by intelligent memory mapping. Previous research overwhelmingly focuses on the former.

In this chapter, we first present an alternative approach to reducing inter-application interference in the memory system: *application-aware memory channel partitioning (MCP)* [9, 10]. The idea is to map the data of applications that are likely to harmfully interfere with each other to different memory channels. The key principles are to partition the data of 1) light (memory non-intensive) and heavy (memory intensive) applications, and of 2) applications with low and high row-buffer locality onto separate channels, respectively. Second, we observe that interference can be even further reduced with a combination of MCP and memory scheduling, which we call *integrated memory partitioning and scheduling (IMPS)*. The key idea is to 1) always prioritize very light applications in the memory scheduler since such applications cause negligible interference to others, 2) use memory channel partitioning to reduce interference between the remaining applications. Extensive evaluations on a variety of multi-programmed workloads and system configurations show that this integrated memory partitioning and scheduling approach provides better system performance than MCP and four previous memory scheduling algorithms employed alone.

In Chapter 4, we address shared cache contention. In addition to continued increase in number of on-chip cores, technology scaling has resulted in not only bigger on-chip caches, but also increase in the number of levels of cache. Modern multicore systems have hierarchical caches, with different subsets of cores exhibiting different degrees of sharing. For instance, a Dell R900 [6] server rack contains two Intel Xeon 7400 [5] chips making it twelve cores with three levels of cache topology , L1, L2 and L3 with varying degrees of sharing among different subsets of cores. In this work, we aim to mitigate the shared cache contention between different applications by intelligently mapping the applications to cores. More specifically, we address the problem of mapping the applications to cores in a multicore system, in the presence of varying degrees of cache sharing among different subsets of cores due to a hierarchical cache structure. The goal of this endeavor is to mitigate contention and improve the system throughput. We use the reuse distance analysis of individual applications to characterize their application behavior and then make mapping decisions.

In Chapter 5, we explore the last-level cache space management problem when the contending threads belong to the same application, as opposed to different applications. In essence, we study the *intra-application* cache partitioning problem [11, 12]. When contending threads belong to the same application, improving the throughput or fairness metrics does not necessarily improve the application performance. This is because, most shared-memory parallel applications contain one or more parallel sections, and the overall performance of a parallel section and consequently, the whole application is determined by the slowest thread, also termed the *critical path thread.* A thread with excellent cache behavior does little to speed up the application performance if the other threads exhibit poor cache behavior. We study the cache behavior of application threads, the positive and negative impact of cache sharing/interference, and, the cache sensitivities of these threads. We later propose a cache partitioning scheme, that dynamically partitions the cache in order to speed up the slower threads by assigning more cache space to them. We maintain runtime performance models of individual threads and employ curve-fitting in order to make cache partitioning decisions. We show that, partitioning the cache this way, speeds up the slower threads, decreases the slack time, and ultimately, improves the overall application performance.

In Chapter 6, we address the problem of efficiently prefetching data into the shared cache when multiple applications are simultaneously performing prefetches. Prefetching is a highly effective latency hiding technique that can greatly improve application performance. However, aggressive prefetching can potentially stress the off-chip bandwidth. The resulting bandwidth stalls can potentially negate the performance gain due to prefetching. In this work [13], focusing on a multicore environment, we first study the comparative benefits of hardware and software prefetching and analyze if the two are complimentary or redundant. This analysis also evaluates different aggressiveness levels of hardware prefetching. Secondly, we weigh the positive performance benefits of prefetching against the negative performance effects of bandwidth stalls. Thirdly, we propose a hierarchical prefetch management scheme for multicores that controls the prefetch levels such that the overall performance gain is improved.

In Chapter 7, we consider the interconnect framework as an instance of a shared on-chip resource. In this work [14], we address the issue of managing the resource efficiently in order to reduce the energy consumption of the shared resource. NoC framework is major contributor to the total on-chip power consumption. The main motivation here is the fact that not all interconnect links are used all the time. We propose to characterize the execution of a multi-threaded application into phases based on the similarity in their inter-core communication pattern. We use this characterization and implement a Markov based prediction scheme, which predicts the link usage pattern of the next execution interval. This prediction is used to proactively turn off predicted unused links and turn on links that are predicted to be used. We show that, managing the link turn-ons and turn-offs in a proactive and dynamic manner reduces the overall energy consumption of the interconnect framework.

In Chapter 8 of the dissertation, we present some possible avenues to explore in future research. We then conclude with a summary of contributions that are made in this dissertation.

Chapter 2

Background

This chapter presents a discussion of the current state-of-the-art in computer architecture and the most likely future trends. We also present requisite background information about memory hierarchy resources.

2.1 Current and Future Trends

Power inefficiency coupled with limited instruction level parallelism changed the trend from increasing single core frequencies to having multiple relatively simpler cores on a single chip. Driven by this need to have power efficient systems, multicore systems have become the order of the day [2] [3] [15] [16]. As technology scales, the number of on-chip cores continues to increase. The future multicore systems are projected to have a large number of relatively simple cores. Although other on-chip resources are expected to increase, their increase is not expected to be commensurate with the increase in number of cores. Therefore, multiple cores are expected to share and contend for these resources. It is also worthwhile noting that, it is efficient for some of these resources to be shared among multiple cores, for instance the last-level cache.

2.2 Shared Resources

The cores in the modern multicore systems share and contend for a number of resources. In this section, we discuss some the most important shared resources. The negative impact of multiple cores contending for a common resource can severely degrade system and application performance if the resource is not efficiently managed. On the flip side, sharing a resource such as the cache can benefit multi-threaded application execution by exploiting potential data sharing between the threads. Therefore, the goal of a resource manager is to mitigate the negative impact of contention and improving metrics such as system throughput, fairness or QoS [17, 18, 19], while at the same time, exploiting the advantages of resource sharing.

2.2.1 On-Chip Cache Hierarchy

Caches and cache hierarchies in CMPs have evolved over the years and span purely private cache organizations, totally shared cache structures and hybrid cache organizations comprising elements of both private and shared cache components. In a purely private cache organization, each core is connected to an L1 cache, which in turn is connected to an L2 cache, both of which are private to the core. Performance isolation, absence of inter-core cache contention and shorter data access times are the main advantages of such purely private cache structures. At the other end of the spectrum is a fully shared L2 cache. In this configuration, each core has a private L1 cache and each of the L1 caches is connected to a common, shared L2 cache. Efficient utilization of available cache space and absence of data redundancy (replication) are the main advantages of this shared cache structure. As CMPs continue to scale and available cache space continues to increase, a purely private cache structure leads to inefficient utilization of cache space and a purely shared cache results in very high contention due to the presence of more cores. This gave rise to hybrid cache architectures comprising elements of both private and shared cache organizations, forming a hierarchical cache structure with multiple levels.

Figure 2.1 shows an eight core system with a three level cache hierarchy. Each core is connected to a private L1 cache and each pair of L1 caches is connected to an L2 cache. Further, in the next level, each pair of L2 caches is connected to an L3 cache. Therefore, there are eight L1, four L2 and two L3 caches in the hierarchy.

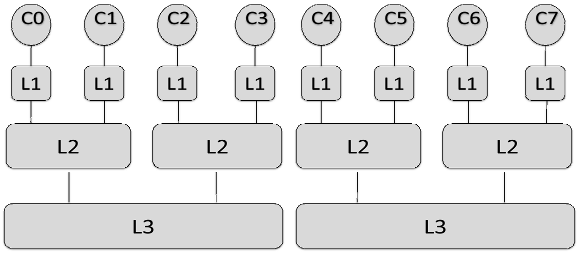


Figure 2.1. A three-level hierarchical cache architecture.

2.2.2 Off-Chip Memory

This section presents a brief background about the DRAM main memory system; more details can be found in [20, 21, 22]. A modern main memory system consists of several channels. Each channel can be accessed independently, i.e., accesses to different channels can proceed in parallel. A memory channel generally refers to an address/command bus and a data bus connected to a dual inline memory module (DIMM). A DIMM has multiple DRAM devices arranged as a rank. A rank of DRAM devices can be accessed in parallel. Figure 2.2 shows eight DRAM devices arranged on a DIMM. In this case, the data bus is 64 bits wide and parallely accesses 8 bits of data from each of the 8 DRAM devices. A DIMM can also have multiple such ra

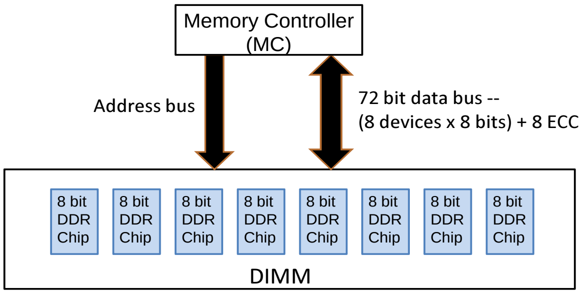


Figure 2.2. A dual inline memory module (DIMM) comprising of eight parallel DRAM devices connected to the memory controller.

A DRAM device is organized as several banks as shown in Figure 2.3. These banks can be accessed in parallel; however, the data and address buses are shared among the banks as mentioned previously, and data from only one bank can be sent through the channel at any time.

Each DRAM bank has a 2D structure consisting of rows and columns. A column is the smallest addressable unit of memory, and a large number of columns make up a row. When a unit of data has to be accessed from a bank, the row containing the data is brought into a small internal buffer called the *row buffer*. If subsequent memory access requests are to the same row, they can be serviced faster (2-3 times) than accessing a new row. This is called a row-hit.

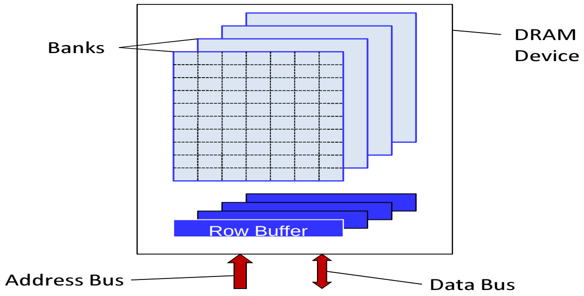


Figure 2.3. A DRAM device with four banks of memory arrays.

A memory controller (MC), which in modern multicore systems is integrated on-chip, is the interface between the processor and the DRAM memory (DIMM) as shown in Figures 2.2 and 2.3. An MC manages the job of scheduling memory requests from the last level cache on the memory channel. In order to improve DRAM data throughput, modern memory controller scheduling algorithms prioritize row-hits over row-misses.

Memory Request Scheduling Policy. FR-FCFS [22, 23] is a commonly used scheduling policy in current commodity systems. It prioritizes row-hits over row-misses, and within each category, it prioritizes older requests. We consider FR-FCFS as the scheduling policy in our analyses. However, in Chapter 3, we evaluate our proposed mechanism with other scheduling policies as well.

OS Page Mapping Policy. The Operating System (OS) maps a virtual address to a physical address. The address interleaving policy implemented in the memory controller in turn maps the physical address to a specific channel/bank in the main memory. Row interleaving and cache line interleaving are two commonly used interleaving policies. In the row interleaving policy, consecutive rows of memory are mapped to consecutive memory channels. We assume equal sizes for OS pages and DRAM rows in this dissertation and use the terms page and row interchangeably without loss of generality. Pure cache line interleaving maps consecutive cache lines in physical address space to consecutive memory channels. A restricted version of cache line interleaving maps consecutive cache lines of a page to banks within a channel.

Commonly used OS page mapping and address interleaving policies are application unaware and map applications' pages across different channels. The OS does not consider inter-application interference and channel information while mapping a virtual page to a physical page. It simply uses the next physical page to allocate/replace based on recency of use. We build our discussions, insights and mechanisms assuming such an interference-unaware OS page mapping policy and a row interleaved address mapping policy.

2.2.3 Interconnect Network

The bus structure acts as an efficient communication fabric when the number of communicating cores is low. With the projected increase in the number of cores in CMPs, limited scalability of bus structures and the need for more on-chip communication bandwidth have become major issues. These issues have given rise to network-on-chip (NoC) [24] [25] [26], which is a more scalable on-chip communication fabric. The NoC framework addresses the scalability issue effectively. However, in such an NoC based CMP, the issue of power consumption can become a serious limiting factor. This is especially true since the power consumption is projected to increase rapidly as the size of NoCs increase. Therefore, there is a need to develop a wide variety of techniques to reduce chip power consumption.

Chapter 3

Application-Aware Memory Channel Partitioning

3.1 Introduction

Applications executing concurrently on a multicore chip contend with each other to access main memory, which has limited bandwidth. If the limited memory bandwidth is not managed well, different applications can harmfully interfere with each other, which can result in significant degradation in both system performance and individual application performance [7, 8, 27, 21, 28, 29]. Several techniques to improve system performance by reducing memory interference among applications have been proposed [7, 8, 27, 21, 28, 29, 30]. Fundamentally, these proposals viewed the problem as a memory access scheduling problem, and consequently focused on developing new memory request scheduling policies that prioritize the requests of different applications in a way that reduces inter-application interference. However, such application-aware scheduling algorithms require (non-negligible) changes to the existing memory controllers' scheduling logic [8, 31].

In this chapter, we present and explore a fundamentally-different alternative approach to reducing inter-application interference in the memory system: controlling the mapping of applications' data to memory channels. Our approach is based on the observation that multicore systems have multiple main memory channels [32, 33, 7] each of which controls a disjoint portion of physical memory and can be accessed independently without any interference [7]. This reveals an interesting trade-off. On the one hand, interference between applications could (theoretically) be completely eliminated if each application's accesses were mapped to a different channel, assuming there were enough channels in the system. But, on the other hand, even if so many channels were available, mapping each application to its own channel would under utilize memory bandwidth and capacity (some applications may need less bandwidth/capacity than they are assigned, while others need more) and would reduce the opportunity for bank/channel-level parallelism within each application's memory access stream. Therefore, the main idea of our approach is to find a sweet spot in this trade-off by *mapping the data (i.e., memory pages) of applications that are likely to cause significant interference/slowdown to each other to different memory channels.*

We make two major contributions. First, we explore the potential of reducing inter-application memory interference purely with channel partitioning, without modifying the memory scheduler. To this end, we develop a new **Application-Aware Memory Channel Partitioning** (MCP) algorithm that assigns preferred memory channels to different applications. The goal is to assign any two applications whose mutual interference would cause significant slowdowns, to different channels. Our algorithm operates using a set of heuristics which are guided by insight about how applications with different memory access characteristics interfere with each other. Specifically, we show in Sec 3.3 that, whenever possible, applications of largely divergent memory-intensity or row-buffer-hit rate should be separated onto different channels.

Second, we show that MCP and traditional memory scheduling approaches are orthogonal in the sense that both concepts can beneficially be applied together. Specifically, whereas our MCP algorithm is agnostic to the memory scheduler (i.e., we assume an unmodified, commonly used row-hit-first memory scheduler [22, 23]), we show that additional gains are possible when using MCP in combination with a minimal-complexity application-aware memory scheduling policy. We develop an **Integrated Memory Partitioning and Scheduling** (IMPS) algorithm that seamlessly divides the work of reducing inter-application interference between the memory scheduler and the system software's page mapper based on what each can do best.

The key insight underlying the design of IMPS is that interference suffered by very low memory-intensity applications is more easily mitigated by prioritizing them in the memory scheduler, than with channel partitioning. Since such applications seldom generate requests, prioritizing their requests does not cause significant interference to other applications, as previous work has also observed [7, 8]. Furthermore, explicitly allocating one or more channels for such applications can result in a waste of bandwidth. Therefore, IMPS prioritizes requests from such applications in the memory scheduler, without assigning them dedicated channels, while reducing interference between all other applications using channel partitioning.

Overview of Results: We evaluate MCP and IMPS on a wide variety of multi-programmed applications and systems and in comparison to a variety of pure memory scheduling algorithms. Our first main finding is that on a 24-core 4-memory controller system with an existing application-unaware memory scheduler, MCP provides slightly higher performance benefits than the best previous memory scheduling algorithm, Thread Cluster Memory Scheduling (TCM) [8]: 7.1% performance improvement vs. 6.1% for TCM. This performance improvement is achieved with no modification to the underlying scheduling policy. Furthermore, we find that IMPS provides better system performance than current state-of-the-art memory scheduling policies, pure MCP, as well as combinations of MCP and state-of-the-art scheduling policies: 5% over the best scheduler, while requiring smaller hardware complexity.

Our main conclusion is that *the task of reducing harmful inter-application memory interference should be divided between the memory scheduler and the system software page mapper.* Only the respective contributions of both entities yields the best system performance.

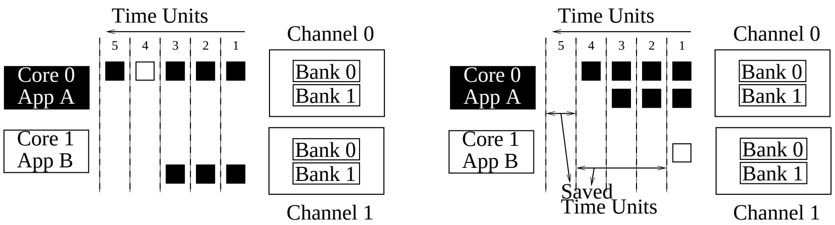
3.2 Background

The background information about DRAM main memory system and the memory request scheduling policy is presented in Section 2.2.2 of Chapter 2. The analyses in this chapter assume the FR-FCFS scheduling policy, but our insights are applicable to other scheduling policies as well. Sec 3.7 describes other memory scheduling policies and Sec 3.8 qualitatively and quantitatively compares our approach to them. The default OS page mapping policy is introduced in Section 2.2.2 of Chapter 2. We build our discussions, insights and mechanisms assuming such an interference-unaware OS page mapping policy and a row interleaved address mapping policy. However, we also evaluate MCP on top of cache line interleaving across banks in Sec 3.9.4.

Memory Related Application Characteristics. We characterize memory access behavior of applications using two attributes. *Memory Access Intensity* is defined as the rate at which an application misses in the last level on-chip cache and accesses memory – calculated as *Misses per Kilo Instructions* (MPKI). *Row Buffer Locality* is defined as the fraction of an application's accesses that hit in the row buffer (i.e., access to an open row). This is calculated as the average *Row-Buffer Hit Rate* (RBH) across all banks.

3.3 Motivation

In this section, we motivate our partitioning approach by showing how applications with certain characteristics cause more interference to other applications, and how careful mapping of application pages to memory channels can ameliorate this problem.



(a) Conventional Page Mapping. (b) Channel Partitioning.

Figure 3.1. Conceptual example showing benefits of mapping data of low and high memory-intensity applications to separate channels.

In Figure 3.1, we present a conceptual example showing the performance benefits of mapping the pages of applications with largely different memory-intensities to separate channels. Application A on Core 0 has high memory-intensity, generating memory requests at a high rate; Application B on Core 1 has low memory-intensity and generates requests at a much lower rate. Figures 3.1(a) and 3.1(b) show characteristic examples of what can happen with conventional page mapping (where the pages of A and B are mapped to the same channels) and with application-aware channel partitioning (where A and B's pages are mapped to separate channels), respectively. In the first case, B's single request is queued up behind 3 of A's requests in a bank of Channel 0 (see Fig 3.1(a)). As a result, Application B stalls for a long period of time (4 DRAM bank access latencies, in this example) until the 3 previously scheduled requests from A to the same bank get serviced. In contrast, if the two applications' data are mapped to separate channels as shown in Figure 3.1(b), B's request is not queued and can be serviced immediately, leading to B's fast progress (1 access latency vs 4 access latencies). Furthermore, even Application A's access latency improves (4 vs. 5 time units) because the interference caused to it by B's single request is eliminated. To determine to what extent such effects occur in practice, we ran a large number of simulation experiments[1] with applications of vastly different memory-intensities and present a representative result: We run four copies each of *milc* and *h264* (from the SPEC CPU2006 suite [34]) on an eight-core, two-channel system.

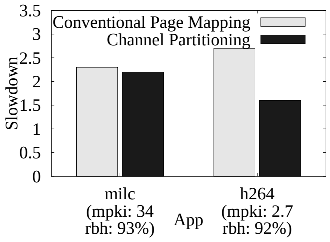


Figure 3.2. Application slowdowns due to interference between high and low memory-intensity applications.

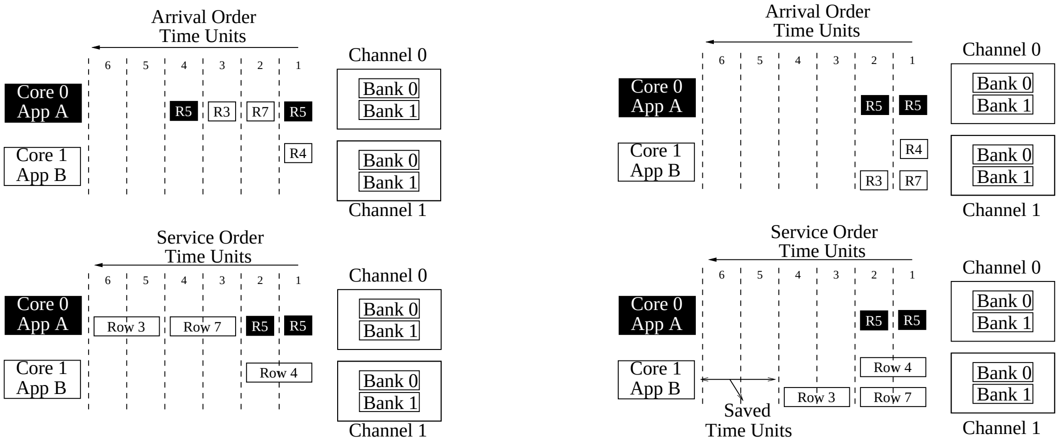
Figure 3.2 shows the effects of conventional channel sharing: *h264*, the application with lower memory-intensity, is slowed down by 2.7x when sharing memory channels with *milc*. On the other hand, if *milc*'s and *h264*'s data are partitioned and mapped to Channels 0 and 1, respectively, *h264*'s slowdown reduces to 1.5x. Furthermore, *milc*'s slowdown also drops from 2.3x to 2.1x, as its queueing delays reduce due to reduced interference from *h264*. This substantiates our intuition from the example: *Separating the data of low memory-intensity applications from that of the high memory-intensity applications can largely improve the performance*

---

[1] Our simulation methodology is described in Sec 3.8.

of both the low memory-intensity applications and the overall system.

Memory-intensity is not the only characteristic that determines the relative harmfulness of applications. In Figure 3.3, we show potential benefits of mapping memory-intensive applications with significantly different row-buffer localities onto separate channels. In the example, Application A accesses the same row, Row 5, repeatedly and hence has much higher row-buffer locality than Application B, whose accesses are to different rows, incurring many row misses.



(a) Conventional Page Mapping (b) Channel Partitioning

Figure 3.3. Conceptual example showing benefits of mapping data of low and high row-buffer hit rate memory-intensive applications to separate channels. In both (a) and (b), the top part shows the request arrival order and the bottom part shows the order in which the requests are serviced.

Figure 3.3(a) shows a conventional page mapping approach, while Figure 3.3(b) shows a channel partitioning approach. With conventional mapping, the commonly used row-hit-first memory scheduling policy prioritizes A's requests over B's requests to Rows 7 and 3, even though B's requests had arrived earlier (Figure 3.3(a)). This leads to increased queueing delays of B's requests causing B to slow down. On the other hand, if the pages of A and B are mapped to separate channels (Figure 3.3(b)), the interference received by B is reduced and consequently the queueing delays experienced by B's requests reduced (by 2 time units). This improves Application B's performance without affecting Application A's.

A representative case study from among our experiments is shown in Figure 3.4. We ran four copies each of *mcf* and *libquantum*, two memory-intensive applications on an eight-core two-channel system. *Mcf* has a low row-buffer hit rate of 42%

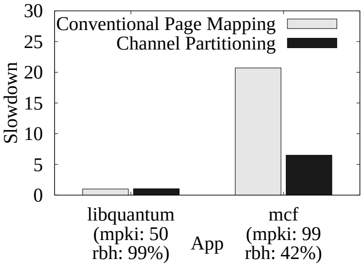


Figure 3.4. Application slowdowns due to interference between high and low row-buffer hit rate memory-intensive applications.

and suffers a slow down of 20.7x when sharing memory channels with *libquantum*, which is a streaming application with 99% row-buffer hit rate. On the other hand, if the data of *mcf* is isolated from *libquantum*'s data and given a separate channel, mcf's slowdown drops significantly, to 6.5x from 20.7x. *Libquantum*'s small performance loss of 4% shows the trade-off involved in channel partitioning: The drop is due to the loss in bank-level parallelism resulting from assigning only one channel to libquantum. In terms of system performance, however, this drop is far outweighed by the reduction in slowdown of *mcf*. We therefore conclude that *isolating applications with low row-buffer locality from applications with high row-buffer locality by means of channel partitioning improves the performance of applications with low row-buffer locality and the overall system.*

Based on these insights, we next develop MCP, an OS-level mechanism to partition the main memory bandwidth across the different applications running on a system. Then, we examine how to best combine memory partitioning and scheduling to minimize inter-application interference and obtain better system performance.

3.4 Memory Channel Partitioning (MCP)

Our MCP mechanism consists of three components: 1) profiling of application behavior during run time, 2) assignment of preferred channels to applications, 3) allocation of pages to the preferred channel. The mechanism proceeds in periodic intervals. During each interval, application behavior is profiled (Sec 3.4.1). At the end of an interval, the applications are categorized into groups based on the characteristics collected during the interval, and each application is accordingly assigned a *preferred channel* (Sec 3.4.2). In the subsequent interval, these preferred channel assignments are applied. That is, when an application accesses a new page that is either not currently in DRAM or not in the application's preferred channel, MCP uses the *preferred channel assignment* for that application: The requested page is allocated in the preferred channel, or migrated to the preferred channel (see Sec 3.4.3).

In summary, during the Xth interval, MCP applies the preferred channel assignment which was computed at the end of the $(X-1)$st interval, and also collects statistics, which will then be used to compute the new preferred channel assignment to be applied during the $(X+1)$st execution interval.[2] Note that MCP does not constrain the memory usage of applications. It provides a preferred channel assignment in order to reduce interference. Therefore, applications can use the entire DRAM capacity, if needed.

3.4.1 Profiling of Application Characteristics

As shown in Sec 3.3, memory access intensity and row-buffer locality are key factors determining the level of harm caused by interference between applications. Therefore, during every execution interval, each application's Misses Per Kilo Instruction (MPKI) and Row-Buffer Hit Rate (RBH) statistics are collected. To compute an application's inherent row-buffer hit rate, we use a per-core shadow row-buffer index for each bank, as in previous work [35, 8, 21], which keeps track of the row that would have been present in the row-buffer had the application been running alone.

3.4.2 Preferred Channel Assignment

At the end of every execution interval, each application is assigned a preferred channel. The assignment algorithm is based on the insights derived in Sec 3.3. The goal is to separate as much as possible 1) the data of low memory-intensity

---

[2] The very first interval is used for profiling only. We envision it to be shorter than the subsequent execution intervals, and its length is a trade off between minimizing the number of pages that get allocated before the first set of channel preferences are established and letting the application's memory access behavior stabilize before collecting statistics. (Empirical evaluation in Sec 3.9.6)

applications from that of high memory-intensity applications, and, 2) among the memory-intensive applications, the data of low row-buffer locality applications from that of high row-buffer locality applications. To do so, MCP executes the following steps in order:

1. Categorize applications into low and high memory-intensity groups based on their MPKI. (Sec 3.4.2.1)

2. Further categorize the high memory-intensity applications, based on their row-buffer hit rate (RBH) values into low and high row-buffer hit rate groups. (Sec 3.4.2.2)

3. Partition the available memory channels among the three application groups. (Sec 3.4.2.3)

4. For each application group, partition the set of channels allocated to this group between all the applications in that group, and assign a preferred channel to each application. (Sec 3.4.3)

3.4.2.1 Intensity Based Grouping

MCP categorizes applications into low and high memory-intensity groups based on a threshold parameter, $MPKI_t$. $MPKI_t$ is determined by averaging the last level cache MPKI of all applications and multiplying it by a scaling factor. For every application i, if its MPKI, $MPKI_i$ is less than $MPKI_t$, the application is categorized as low memory-intensity, else high memory-intensity. The average value of MPKI provides a threshold that adapts to the workload's memory intensity and acts as a separation point in the middle of the workload's memory-intensity range. The scaling factor further helps to move this point up or down the range, regulating the number of applications in the low memory-intensity group. We empirically found that a scaling factor of 1 provides an effective separation point and good system performance (Sec 5.5).

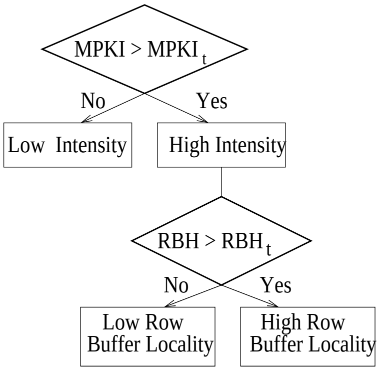


Figure 3.5. MCP: Application Grouping.

3.4.2.2 Row-Buffer Locality Based Grouping

MCP further classifies the high memory-intensity applications into either low or high row-buffer locality groups based on a threshold parameter, RBH_t. For every application i, if its RBH_i is less than RBH_t, then it is classified as a low row-buffer locality application. In this case, we do not take an average or use a scaling factor, as we observe that inter-application interference due to row-buffer locality differences are more pronounced between applications with very low and high row-buffer localities, unlike memory-intensity where there is interference across the continuum. We empirically observe that an RBH_t value of 50% provides effective interference reduction and good performance (Sec 5.5).

3.4.2.3 Partitioning Channels between Application Groups

After thus categorizing applications into 3 groups, MCP partitions the available memory channels between the groups. It is important to note that at this stage of the algorithm, memory channels are assigned to application groups and *not* to individual applications. MCP handles the preferred channel assignment to individual applications in the next step (Sec 3.4.2.4). Channels are first partitioned between low and high memory-intensity groups. The main question is how many channels should be assigned to each group. One possibility is to allocate channels proportional to the total bandwidth demand (sum of applications' MPKI) of each group (bandwidth proportional allocation). This amounts to balancing the total bandwidth demand across channels. Alternatively, channels could be allocated proportional to the number of applications in that group (application count proportional allocation). In the former case, the low memory-intensity applications which constitute a very low proportion of total bandwidth demand might be assigned no channels. This fails to achieve their isolation from high memory-intensity applications, leading to low system performance. In contrast, if the latter is used, it results in bandwidth wastage as the low memory-intensity applications seldom generate requests and the bandwidth of the channels they are assigned to would have been better utilized by the high memory-intensity applications. We found that the isolation benefits of application-count-proportional allocation outweighs the bandwidth wastage caused by potentially allocating low-intensity applications to one or more channels.[3] Therefore, we use the application count proportional channel allocation strategy for MCP. However, bandwidth wastage caused by potentially allocating very low intensity applications dedicated channels remains, and we will show that eliminating this wastage by handling these applications in the scheduler in an integrated scheduling and partitioning mechanism is beneficial (Sec 3.5). The channels allocated to the high memory-intensity group are further partitioned between the low and high row-buffer locality groups. The applications in the high memory-intensity group are bandwidth sensitive, meaning they each need a fair share of bandwidth to make progress. To ensure this, MCP assigns a number of channels to each of these two groups proportionally to the bandwidth demand (sum of MPKIs) of the group.

3.4.2.4 Preferred Channel Assignment within an Application Group

As a final step, MCP determines which applications within a group are mapped to which channels, when more than one channel is allocated to a group. Within each group, we balance the total bandwidth demand across the allocated channels. For each group, we maintain a ranking of applications by memory-intensity. We start with the least intensive application in the group and map applications to the group's first allocated channel until the bandwidth demand allocated to it (approximated by sum of $MPKI_i$ of every application i allocated to it) is $\frac{\text{Sum of MPKIs of applications in the group}}{\text{Number of channels allocated to the group}}$. We then move on to the next channel and allocate applications to it. This is repeated for every application group. At the end of this procedure, each application is assigned a *preferred channel*.

3.4.3 Allocation of Pages to Preferred Channel

Once each application is assigned a preferred channel, MCP allocates a page to the preferred channel in case it is not already there. There are two possibilities. First, a page fault: the accessed page is not present in any channel. In this case, the page fault handler attempts to allocate the page in the preferred channel. If there is a free page in the preferred channel, the new page is allocated there. Otherwise,

---

[3] We found that bandwidth proportional allocation results in a 4% performance loss over the baseline since it increases memory interference.

a modified version of the CLOCK replacement policy, as described in [36] is used. The baseline CLOCK policy keeps a circular list of pages in memory, with the hand (iterator) pointing to the oldest allocated page in the list. There is a Referenced (R) bit for each page, that is set to '1' when the page is referenced. The R bits of all pages are cleared periodically by the operating system. When a page fault occurs and there are no free pages, the hand moves over the circular list until an unreferenced page (a page with R bit set to '0') is found. The goal is to choose the first unreferenced page as the replacement. To allocate a page in the preferred channel, the modified CLOCK algorithm looks ahead N pages beyond the first replacement candidate to potentially find an unreferenced page in the preferred channel. If there is no unreferenced page within N, the first unreferenced page in the list across all channels is chosen as the replacement candidate. We use an N value of 512.

Second, the accessed page is present in a channel other than the preferred channel, which we observe to be very rare in our workloads, since application behavior is relatively constant within an interval. In this case, dynamically migrating the page to the preferred channel could be beneficial. However, dynamic page migration incurs TLB and cache block invalidation overheads as discussed in [1]. We find that less than 12% of pages in all our workloads go to non-preferred channels and hence migration does not gain much performance over allowing some pages of an application to potentially remain in the non-preferred channels. Thus, our default implementation of MCP does not do migrations. However, if needed, migration can of course be seamlessly incorporated into MCP and IMPS.

3.5 Integrated Partitioning/Scheduling (IMPS)

MCP aims to solve the inter-application memory interference problem entirely with the system software's page mapper (with the support of additional hardware counters to collect MPKI and RBH metrics for each application). It does not require any changes to the memory scheduling policy. This approach is in stark contrast to the various existing proposals, which try to solve the problem "from the opposite side". These proposals aim to reduce memory interference entirely in the memory controller hardware using sophisticated scheduling policies (e.g., [7, 8, 21, 28]) The question is whether either extreme alone (i.e., page mapping alone and memory scheduling alone) can really provide the best possible interference reduction. Based on our observations, the answer is negative. Specifically, we devise an integrated memory partitioning and scheduling (IMPS) mechanism that aims to combine the interference reduction benefits of both.

The key observation underlying IMPS is that applications with very low memory intensity, when prioritized over other applications in the memory scheduler, do not cause significant slowdowns to other applications. This observation was also made in previous work [7, 8]. These applications seldom generate memory requests; prioritizing these requests enables the applications to quickly continue with long computation periods and utilize their cores better, thereby significantly improving system throughput [7, 8]. As such, scheduling can very efficiently reduce interference that affects very low memory-intensity applications. In contrast, reducing the interference against such applications purely using the page mapper is inefficient. The mapper would have to dedicate one or more channels to such low-memory-intensity applications, wasting memory bandwidth, since these applications do not require significant memory bandwidth (yet high-intensity applications would likely need the wasted bandwidth, but cannot use it). If the mapper cannot dedicate a channel to such applications, they would share channels with high-intensity applications and experience high interference with an unmodified memory scheduler.

The basic idea and operation of IMPS is therefore simple. First, identify at the end of an execution interval very low memory-intensity applications (i.e., applications whose MPKI is smaller than a very low threshold, 1.5 in most of our experiments (Sec 3.9.6)), prioritize them in the memory scheduler over all other applications in the next interval, and allow the mapping of the pages of such applications to any memory channel. Second, reduce interference between all other applications by using memory channel partitioning (MCP), exactly as described in Sec 3.4. The modification to the memory scheduler is minimal: the scheduler only distinguishes the requests of very low memory-intensity applications over those of others, but does not distinguish between requests of individual applications in either group. The memory scheduling policy consists of three prioritization rules: 1) prioritize requests of very low memory-intensity applications, 2) prioritize row-hit-first requests, 3) prioritize older requests.

Note that MCP is still used to classify the remaining applications as low and high memory-intensity, as only the very low memory-intensity applications are filtered out and prioritized in the scheduler. MCP's channel partitioning still reduces interference and consequent slowdowns of the remaining applications.

3.6 Implementation

Hardware support. MCP requires hardware support to estimate MPKI and row-buffer hit rate of each application, as described in Sec 3.4.1. These counters are readable by the system software via special instructions. Table 3.1 shows the storage cost incurred for this purpose. For a 24-core system with 4 memory controllers (each controlling 4 memory banks and 16384 rows per bank), the hardware overhead is 12K bits. IMPS requires an additional bit per each request (called *low-intensity bit*) to distinguish very low-memory-intensity applications' requests over others, which is an additional overhead of only 512 bits for a request queue size of 128 per MC. IMPS also requires small modifications to the memory scheduler to take into account the *low-intensity bits* in prioritization decisions. Note that, unlike previous application-aware memory request scheduling policies, IMPS (or MCP) 1) does not require each main memory request to be tagged with a thread/application ID since it does not distinguish between individual applications' requests, 2) adds only a single new bit per request for the memory scheduler to consider, 3) does not require application ranking as in [7, 8, 28] – ranking and prioritization require hardware logic for sorting and performing comparisons. As such, the complexity of IMPS is much lower than previous application-aware memory scheduling policies.

Storage	**Description**	**Size**
Storage Overhead for MCP - per-core registers		
MPKI-counter	A core's last level cache misses per kilo instruction	$N_{core} \times log_2 MPKI_{max} = 240$
Storage Overhead for MCP - per-core registers in each controller		
Shadow row-buffers	Row address of a core's last accessed row	$N_{core} \times N_{banks} \times log_2 N_{rows} = 1344$
Shadow row-buffer hit counters	Number of row-hits if the application were running alone	$N_{core} \times N_{banks} \times log_2 Count_{max} = 1536$
Additional Overhead for IMPS - per request register in each MC		
Very low memory -intensity indicator	To identify requests from very low memory-intensity applications	$1 \times Queue_{max} = 128$

Table 3.1. Hardware storage required for MCP and IMPS

System software support. MCP and IMPS require support from system software to 1) read the counters provided by the hardware, 2) perform the preferred channel assignment, at the end of each execution interval, as already described. Each application's preferred channel is stored as part of the system software's data structures, leading to a very modest memory overhead of $N_{AppsInSystem} \times N_{MemoryChannels}$. The page fault handler and the page replacement policy are modified slightly, as described in Sec 3.4.3. Our experiments show that the execution time overheads of the required tasks are negligible. Note that our proposed mechanisms do not require changes to the page table.

3.7 Related Work and Qualitative Comparisons to Previous Work

To our knowledge, this is the first work to propose and explore memory page mapping mechanisms as a solution to mitigate inter-application memory interference and thereby improve system performance.

Memory Scheduling. The problem of mitigating interference has been extensively addressed using application-aware memory request scheduling. We briefly describe the two approaches we compare our mechanisms to in Section 5.5. ATLAS [7] is a memory scheduling algorithm that improves system throughput by prioritizing applications based on their attained memory service. Applications that have smaller attained memory service are prioritized over others because such threads are more likely to return to long compute periods and keep their cores utilized. Thread cluster memory scheduling (TCM) [8] improves both system performance and fairness. System performance is improved by allocating a share of the main memory bandwidth for latency-sensitive applications. Fairness is achieved by shuffling scheduling priorities of memory-intensive applications at regular intervals to prevent starvation of any application. These works and other application-aware memory scheduler works [27, 21, 28, 29, 30] attempt to reduce inter-application memory interference purely through memory scheduling. As a result, they require significant modifications to the memory controller's design. In contrast, we propose 1) an alternative approach to reduce memory interference which does not require changes to the scheduling algorithm when employed alone, 2) combining our channel partitioning mechanism with memory scheduling to gain better performance than either can achieve alone. Our quantitative comparisons in Section 5.5 show that our proposed mechanisms perform better than the current state-of-the-art scheduling policies, with no change or minimal changes to the memory scheduling algorithm.

Application-unaware memory schedulers [37, 38, 22, 23], including the commonly employed FR-FCFS policy [22, 23], aim to maximize DRAM throughput, and therefore, lead to low system performance in multi-core systems, as shown in previous work [7, 8, 27, 21, 28, 29].

OS Thread Scheduling. Zhuravlev et al. [39] aims to mitigate shared resource contention between threads by co-scheduling threads that interact well with each other on cores sharing the resource, similar to [40]. Such solutions require enough threads with symbiotic characteristics to exist in the OS's thread scheduling pool. In contrast, our proposal can reduce memory interference even if threads that interfere significantly with each other are co-scheduled in different cores and can be combined with co-scheduling proposals to further improve system performance.

Page Allocation. Page allocation mechanisms have been explored previously. Awasthi et al. [1] use page allocation/migration to balance load across memory controllers (MCs) in an application-unaware manner, to improve memory bandwidth utilization and system performance in a network-on-chip based system where a core has different distances to different memory channels. Our proposal, in comparison, performs page allocation in an application-aware manner with the aim of reducing interference between different applications. We compare our approach to an adaptation of [1] to crossbar-based multicore systems where all memory controllers are equidistant to any core (in Section 3.9.3) and show that application-aware channel partitioning leads to better system performance than balancing load in MCs. However, concepts from both approaches can be combined for further performance benefits. In NUMA-based multiprocessor systems with local and remote memories, page allocation mechanisms were used to place data close to corresponding computation node [41, 42]. Our goal is completely different: to map data to different channels to mitigate interference between different applications. In fact, our schemes do not require the system to have non-uniform access characteristics to MCs.

Sudan et al. [43] propose to colocate frequently used chunks of data into rows, thereby improving row-buffer locality, by modifying OS page mapping mechanisms. Lebeck et al. and Hur et al. [37, 44] propose page allocation mechanisms to increase idleness and thus decrease energy consumption in DRAM ranks/banks. Phadke et al. [45] propose a heterogeneous memory system where each memory channel is optimized for latency, bandwidth, or power and propose page mapping mechanisms to map appropriate applications' data to appropriate channels to improve performance and energy efficiency. None of these works consider using page allocation to reduce inter-application memory interference, and therefore they can be potentially combined with our proposal to achieve multiple different goals.

3.8 Evaluation Methodology

Simulation Setup. MCP requires the MPKI and RBH values to be collected for each application. These per-application hardware counters, though easy to implement, are not present in existing systems. Also, our evaluation requires different system configurations with varying architectural parameters and comparison to new scheduling algorithms. For these reasons, we are unable to evaluate MCP on a real system and use an in-house cycle-level x86 multi-core simulator. The front end of the simulator is based on Pin [46]. This simulator models the memory subsystem of a CMP in detail. It enforces channel, rank, bank, port and bus conflicts, thereby capturing all the bandwidth limitations and modeling both channel and bank-level parallelism accurately. The memory model is based on DDR2 timing parameters [47], verified using DRAMSim [48]. We model the execution in a core, including the instruction-window. Unless mentioned otherwise, we model a 24-core system with 4 memory channels/controllers. Table 6.1 shows major processor and memory parameters.

Evaluation Metrics. We measure the overall throughput of the system using *weighted speedup* [40]. We also report *harmonic speedup*, which is a combined measure of performance and fairness.

Processor Pipeline	128-entry instruction window (64-entry issue queue, 64-entry store queue), 12-stage pipeline
Fetch/Exec/Commit Width	3 instructions per cycle in each core; 1 can be a memory operation
L1 Caches	32 K-byte per-core, 4-way set associative, 32-byte block size, 2-cycle latency
L2 Caches	512 K-byte per core, 8-way set associative, 32-byte block size, 12-cycle latency
DRAM controller (on-chip)	128-entry request buffer, 64-entry write buffer, reads prioritized over writes, row interleaving
DRAM chip parameters	DDR2-800 timing parameters, t_{CL}=15ns, t_{RCD}=15ns, t_{RP}=15ns, BL/2=10ns, 8 banks, 4K row-buffer
DIMM Configuration	Single-rank, 8 DRAM chips put together on a DIMM to provide a 64-bit wide memory channel
Round-trip L2 miss latency	For a 32-byte cache line, uncontended: row-buffer hit: 40ns (200 cycles), closed: 60ns (300 cycles), conflict: 80ns (400 cycles)
Cores and DRAM controllers	24 cores, 4 independent DRAM controllers, each controlling a single memory channel

Table 3.2. Default processor and memory subsystem configuration.

$$SystemThroughput = WeightedSpeedup = \Sigma_i \frac{IPC_i^{shared}}{IPC_i^{alone}};$$
$$HarmonicSpeedup = \Sigma_i \frac{N}{\frac{IPC_i^{alone}}{IPC_i^{shared}}}.$$

Our normalized results are normalized to the FR-FCFS baseline, unless stated otherwise.

Workloads. We use workloads constructed from the SPEC CPU2006 benchmarks [34] in our evaluations. We compiled the benchmarks using gcc with the O3 optimization flag. Table 3.3 shows benchmarks' characteristics. We classify benchmarks into two categories: high memory-intensity (greater than 10 MPKI) and low memory-intensity (less than 10 MPKI). We vary the fraction of high memory-intensity benchmarks in our workloads from 0%, 25%, 50%, 75%, 100% and construct 40 workloads in each category. Within each memory-intensity category, we vary the fraction of high row-buffer hit rate benchmarks in a workload from low to high. We also create another category, $VeryLow(VL)$ of 40 workloads. All benchmarks in these workloads have less than 1 MPKI. We consider VL for completeness, although these workloads have little bandwidth demand. For our main evaluations and some analyses, we use all 240 workloads and run each workload for 300M cycles. For sensitivity studies, we use the 40 balanced (50% memory-intensive) workloads, unless otherwise mentioned, and run for 100M cycles to reduce simulation time.

Parameter Values. The default MPKI scaling factor and RBH_t values we use in our experiments are 1 and 50% respectively. For the *profile interval* and *execution*

No.	Benchmark	MPKI	RBH	No.	Benchmark	MPKI	RBH
1	453.povray	0.03	85.15%	14	456.hmmer	5.69	35.47%
2	400.perlbench	0.13	83.64%	15	473.astar	9.21	76.17%
3	465.tonto	0.16	91%	16	436.cactusADM	9.37	17.95%
4	454.calculix	0.20	87.2%	17	471.omnetpp	21.61	46%
5	444.namd	0.32	95.4%	18	483.xalancbmk	23.85	73.17%
6	481.wrf	0.33	91.9%	19	482.sphinx3	24.85	85.38%
7	403.gcc	0.37	73.23%	20	459.GemsFDTD	25.30	28.77%
8	458.sjeng	0.42	11.53%	21	433.milc	34.33	93.24%
9	447.dealIII	0.45	81.23%	22	470.lbm	43.52	95.18%
10	445.gobmk	0.62	71.01%	23	462.libquantum	50.06	99.21%
11	435.gromacs	0.73	84.43%	24	450.soplex	50.08	91.25%
12	464.h264	2.70	92.3%	25	437.leslie3d	59.03	82.6%
13	401.bzip2	3.90	53.82%	26	429.mcf	99.79	42.87%

Table 3.3. SPEC CPU2006 benchmark characteristics.

interval, we use values of 10 million and 100 million, respectively. We later study sensitivity to all these parameters.

3.9 Results

We first present and analyze the performance of MCP and IMPS on a 24-core 4-memory controller system. Figure 3.6 shows the system throughput and harmonic speedup averaged over all 240 workloads.

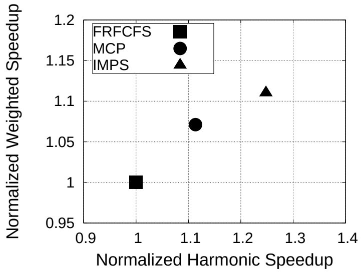


Figure 3.6. MCP and IMPS performance (normalized) across 240 workloads.

The upper right part of the graph corresponds to better system throughput and a better balance between fairness and performance. MCP improves system throughput by 7.1% and harmonic speedup by 11% over the baseline. IMPS provides 4% better system throughput (13% better harmonic speedup) over MCP, and 11% better system throughput (24% better harmonic speedup) over the baseline. We observe (not shown) that the scheduling component of IMPS alone (without partitioning) gains half of the performance improvement of IMPS. We conclude that interference-aware channel partitioning is beneficial for system performance, but dividing the task of interference reduction using both channel partitioning and memory scheduling together provides better system performance than employing either alone.

Individual Workloads. Figure 3.7 shows the weighted speedup for four randomly selected, representative workloads shown in Table 3.4. We observe that MCP and IMPS gain performance benefits consistently across different workloads.

Workload	High memory-intensity benchmarks	Low Memory-intensity benchmarks
W1	perlbench, gobmk, gromacs, gcc(2), sjeng(2), hmmer(2), bzip2, cactus, h264ref	sphinx3, soplex, libquantum(4), milc(3), lbm(3)
W2	gromacs(2), h264(2), dealII(2), astar(2), hmmer(2), cactusADM(2)	milc(2), leslie3d(2)), sphinx(3), gemsFDTD(3), libquantum(3)
W3	namd(3), gcc(3), astar(3), cactusADM(3)	leslie3d(3), milc(3), omnetpp(3), mcf(3)
W4	tonto, astar, gcc, povray, hmmer, h264, bzip2 gromacs, perlbench, dealII, cactusADM(2),	xalancbmk, libquantum, lbm, sphinx3, milc soplex, omnetpp(2), gemsFDTD(3), mcf

Table 3.4. Four representative workloads.

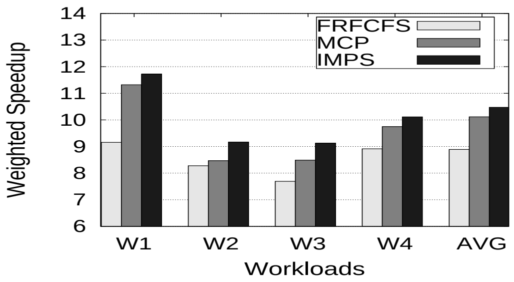


Figure 3.7. MCP and IMPS performance for 4 sample workloads and avg across 40 balanced workloads.

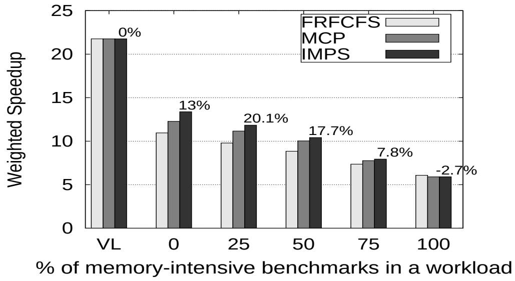


Figure 3.8. MCP and IMPS Performance across memory-intensity categories. % gain values of IMPS over FR-FCFS are labeled.

Effect of Workload Memory-Intensity. Figure 3.8 shows the system throughput benefits of MCP and IMPS, for six memory-intensity based categories of workloads.[4] As expected, as workload intensity increases (from left to right in the figure), absolute system throughput decreases due to increased interference between applications.

We make three major conclusions. First, MCP and IMPS improve performance significantly over FR-FCFS in most of the memory-intensity categories. Specifically, MCP avoids interference between applications of both dissimilar and similar

---

[4] All categories from 0 - 100% place a load on the memory system, as the intensity cut off used to classify an application as intensive is 10 MPKI, which is reasonably large to begin with.

intensities by isolating them to different channels, enabling benefits mostly regardless of workload composition. Second, IMPS's performance benefit over MCP is especially significant in the lower-intensity workloads. Such workloads have a higher number of very low memory-intensity applications and IMPS prioritizes them in the scheduler, which is more effective for system performance than reducing interference for them by assigning them to their own channels, which wastes bandwidth as done by MCP. As the workload memory-intensity increases, IMPS' performance benefit over MCP becomes smaller because the number of low-intensity workloads becomes smaller. Third, when the workload mix is very non-intensive or very intensive, MCP/IMPS do not provide much benefit. In the VL category, load on memory and as a result interference is very low, limiting the potential of MCP/IMPS. When 100% of applications in the workload are intensive, the system becomes memory bandwidth limited and conserving memory bandwidth by exploiting row-buffer locality (using simple FR-FCFS) provides better performance than reducing inter-application interference at the expense of reducing memory throughput. Any scheduling or partitioning scheme that breaks the consecutive row-buffer hits results in a system performance loss. We conclude that MCP and IMPS are effective for a wide variety of workloads where contention exists and the system is not fully bandwidth limited.

3.9.1 Comparison with Previous Scheduling Policies

Figure 3.9 compares MCP and IMPS with previous memory scheduling policies, FR-FCFS [22], PARBS [28], ATLAS [7] and TCM [8] over 240 workloads. Two major conclusions are in order. First, application-aware scheduling policies perform better than FR-FCFS, and, TCM performs the best among the application-aware scheduling policies, consistent with previous work[7, 8, 28]. Second, MCP and IMPS outperform TCM by 1%/5%, with no/minimal changes to the scheduler.

Figure 3.10 provides insight into where MCP and IMPS' performance benefits are coming from by breaking down performance based on workload intensity. As the workload memory intensity (thus contention) increases, MCP and IMPS become more effective than pure memory scheduling approaches. At low-intensity workloads (VL, 0%, 25%), TCM performs slightly better than IMPS because TCM

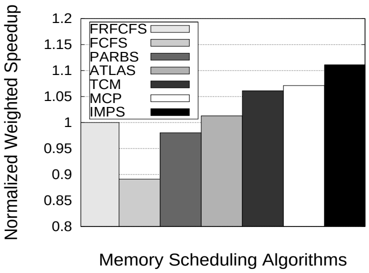


Figure 3.9. MCP and IMPS performance (normalized) vs previous scheduling policies averaged across 240 workloads.

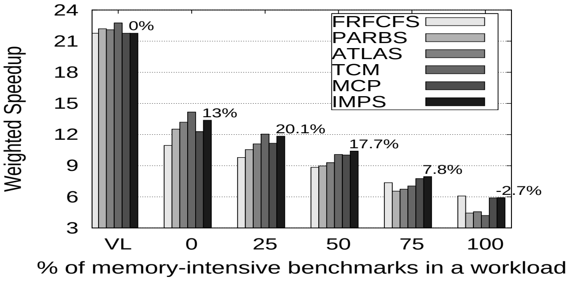


Figure 3.10. MCP and IMPS performance vs previous scheduling policies across memory-intensity categories. Percentage improvement values of IMPS over FR-FCFS are displayed.

is able to distinguish and prioritize between each individual application in the memory scheduler (not true for MCP/IMPS), leading to reduced interference between low and medium intensity applications. At higher intensity workloads (50%, 75%, 100%), reducing interference via channel partitioning is more effective than memory scheduling: both MCP and IMPS outperform TCM, e.g. by 40% in the 100%-intensity workloads. In such workloads, contention for memory is very high as many high-intensity applications contend. Channel partitioning completely eliminates interference between some applications by separating out their access streams to different channels, thereby reducing the number of applications that contend with each other. On the other hand, TCM or a pure memory scheduling scheme tries to handle contention between high-intensity workloads purely by prioritization, which is more effective at balancing interference but cannot eliminate interference as MCP/IMPS does since all applications contend with each other. We conclude that IMPS is a more effective solution than pure memory scheduling especially when workload intensity (i.e., memory load) is high, which is the expected trend in future systems.

Note that IMPS's performance benefits over application-aware memory schedulers come at a significantly reduced complexity, as described in Section 3.6.

3.9.2 Interaction with Previous Scheduling Policies

Figure 3.11 compares MCP and IMPS, when implemented on top of FR-FCFS, ATLAS and TCM as the underlying scheduling policy. When IMPS is implemented over ATLAS and TCM, it adds another priority level on top of the scheduling policy's priority levels: very-low-intensity applications are prioritized over others and the scheduling policy's priorities are used between very-low-intensity applications and between the remaining applications.

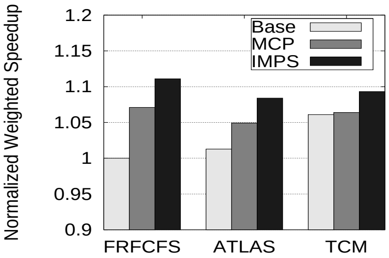


Figure 3.11. MCP and IMPS performance over different scheduling policies (240 workloads).

Several conclusions are in order. First, adding MCP/IMPS on top of any previous scheduling policy improves performance (IMPS gains 7% and 3% over ATLAS and TCM respectively), showing that our proposal is orthogonal to the underlying memory scheduling algorithm. Second, MCP/IMPS over FR-FCFS (our default proposal) provides better performance than MCP/IMPS employed over TCM or ATLAS. This is due to two reasons: 1) channel partitioning decisions MCP makes are designed assuming an FR-FCFS policy and not designed to take into account or interact well with ATLAS/TCM's more sophisticated thread ranking decisions. There is room for improvement if we design a channel partitioning scheme that is specialized for the underlying scheduling policy. We leave this for future work. 2) MCP/IMPS isolates groups of similar applications to different channels and ATLAS/TCM operate within each channel to prioritize between/cluster these similar applications. However, ATLAS and TCM are designed to exploit heterogeneity between applications and do not perform as well when the applications they prioritize between are similar. We found that prioritizing similar-intensity applications over each other in the way ATLAS/TCM does, creates significant slowdowns because the applications are treated very differently. We conclude that MCP/IMPS can be employed on top of any underlying scheduler to gain better performance over using the scheduler alone. However, it performs best when employed over an FR-FCFS baseline for which it is designed.

3.9.3 Comparison with Prior Work on Page Mapping

In [1], Awasthi et al propose two page allocation schemes to balance the load across multiple memory controllers: 1) page allocation on first touch (Adaptive First Touch, AFT), 2) Dynamic Page Migration (DPM). AFT attempts to balance load by allocating a page to a channel which has the minimum value of a cost function involving channel *load*, row buffer hit rate, and, the *distance* to the channel. DPM proposes to migrate a certain number of pages from the channel with the highest load to the least loaded channel at regular intervals, in addition to AFT. In our adaptation of AFT, we consider both channel load and row-buffer-hit rate but do not incorporate the channel distance, as we do not model a network-on-chip. Figure 3.12 compares MCP/IMPS performance to that of AFT and DPM.

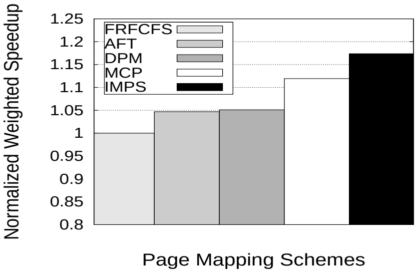


Figure 3.12. MCP and IMPS Performance vs load balancing across memory controllers [1] (40 workloads).

First, AFT and DPM both improve performance by 5% over the baseline, because they reduce memory access latency by balancing load across different channels. The gains from the two schemes are similar as the access patterns of the applications we evaluate do not vary largely with time, resulting in very few invocations of dynamic page migration. Second, our proposals outperform AFT and DPM by 7% (MCP) and 12.4% (IMPS), as they proactively reduce inter-application interference by using application characteristics, while AFT and DPM are not interference- or application-aware and try to reactively balance load across memory controllers. We conclude that reducing inter-application interference by page allocation provides better performance than balancing load across memory controllers in an application-unaware manner.

3.9.4 Impact of Cache Line Interleaving

We study the effect of MCP/IMPS on a system with a restricted form of cache line interleaving that maps consecutive cache lines of a page across banks within a channel. Figure 3.13 shows that MCP/IMPS improve the performance of such a system by 5.1% and 11% respectively. We observed (not shown) that unrestricted cache line interleaving across channels (to which MCP/IMPS cannot be applied) improves performance by only 2% over restricted cache line interleaving. Hence, using channel partitioning with MCP/IMPS outperforms cache line interleaving across channels. This is because the reduction in inter-application interference with MCP/IMPS provides more system performance benefit than the increase of channel-level parallelism with unrestricted cache-line interleaving. We conclude that MCP/IMPS are effective independent of the interleaving policy employed, as long as the interleaving policy allows the mapping of an entire page to a channel (which is required for MCP/IMPS to be implementable).

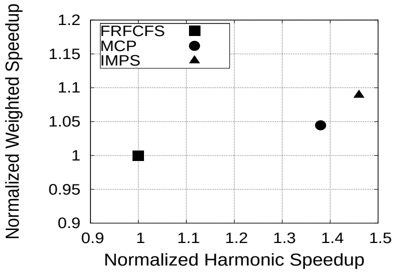


Figure 3.13. System throughput and harmonic speedup with cache line interleaving (240 workloads).

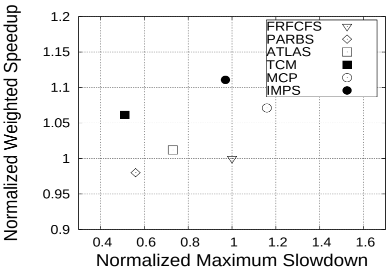


Figure 3.14. Performance and fairness compared to previous scheduling policies (240 workloads).

3.9.5 Effect of MCP and IMPS on Fairness

The fairness metric we use, the maximum slowdown of a workload, is defined as the maximum of the slowdowns (inverse of speedups) of all applications [7, 8, 49]; lower maximum slowdown values are more desirable. Figure 3.14 shows throughput vs fairness of previously proposed scheduling policies and our proposed schemes. IMPS has slightly better fairness (3% lower maximum slowdown) than FR-FCFS. While MCP and IMPS provide the best performance compared to any other previous proposal, they result in higher unfairness. Note that this is expected by design: MCP and IMPS are designed for improving system performance and not fairness. They make the conscious choice of placing high-intensity (and high-row-locality) applications onto the same channel(s) to enable faster progress of lower-intensity applications, which sometimes results in the increased slowdown of higher-intensity applications. Channel partitioning based techniques that can improve both performance and fairness are out of the scope of this chapter and an interesting area of future work.

3.9.6 Sensitivity Studies

Sensitivity to MCP and IMPS algorithm parameters. We first vary the profile interval length to study its impact on MCP and IMPS' performance (Figure 3.15). A shorter initial profile interval length (1 and 5 Million) leads to less stable MPKI and RBH values, leading to inaccurate estimation of application characteristics. In contrast, a longer profile interval length causes a number of pages to be allocated prior to computing channel preferences. A profile interval length of 10M cycles balances these downsides of shorter and larger intervals and provides the best performance. We also experimented with different execution interval lengths (Figure 3.16). A shorter interval leads to better adaptation to changes in application behavior but also higher overhead due to page migration if application characteristics are not stable within the interval. A longer interval might miss changes in the behavior of applications. A 100M-cycle interval ensures a good balance and provides good performance.

Figure 3.17 shows the sensitivity of MCP/IMPS to $MPKI_t$. As $MPKI_t$ is increased beyond 1, more medium and high memory-intensity applications get into the low memory-intensity group, thereby slowing down the low-intensity applications and resulting in lower throughput. We also varied RBH_t, the row buffer-hit rate threshold and the very low memory-intensity threshold. System performance remains high and stable over a wide range of these values, with the best performance observed at an RBH_t value of 50% and a very low memory-intensity threshold value of 1.5.

Scalability to cores, MCs and cache sizes. Table 3.5 shows the performance of IMPS as number of cores, number of MCs and L2 cache size are varied. The

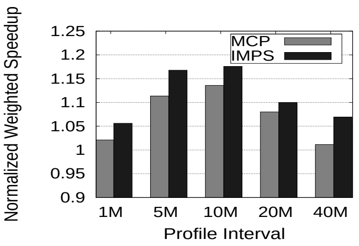


Figure 3.15. Performance vs Profile interval (40 workloads).

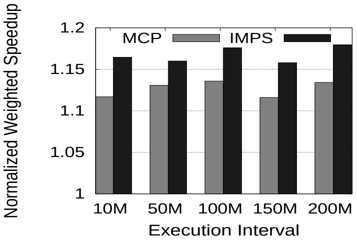


Figure 3.16. Performance vs Execution interval (40 workloads).

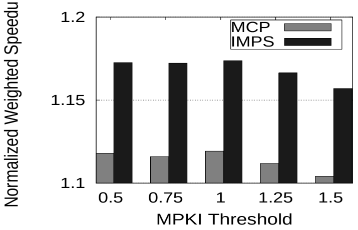


Figure 3.17. Performance vs $MPKI_t$ (40 workloads).

rest of the system remains the same. IMPS' benefits are significant across all configurations. IMPS' performance gain in general increases when the system is more bandwidth constrained, i.e., with increasing number of cores and reducing number of MCs. MCP shows similar trends as IMPS.

	No. of Cores			No. of MCs			Private L2 Cache Size		
	16	24	32	2	4	8	256KB	512KB	1MB
IMPS System Throughput gain	15.8%	17.4%	31%	18.2%	17.1%	10.7%	16.6%	17.4%	14.3%

Table 3.5. Sensitivity to number of cores, number of MCs, and L2 cache size (40 workloads).

3.10 Conclusion

We presented 1) MCP, a fundamentally new approach to reducing inter-application interference at the memory system, by mapping the data of interfering applications to separate channels, 2) IMPS, that effectively divides the work of reducing inter-application interference between the system software and the memory scheduler. Our extensive qualitative and quantitative comparisons demonstrate that MCP and IMPS both provide better system performance than the state-of-the-art memory scheduling policies, with no or minimal hardware complexity. IMPS provides better performance than channel partitioning or memory scheduling alone. We conclude that inter-application memory interference is best reduced using the right combination of page allocation to channels and memory scheduling, and that IMPS achieves this synergy with minimal hardware complexity.

Chapter 4

Reuse Distance Based Performance Modeling and Workload Mapping

4.1 Introduction

Many current chip multiprocessors (CMPs) support on-chip hierarchical caches. For instance, Intel Xeon 7400 processor (previously code-named Intel Dunnington) [5] has six on-chip cores with each pair of cores sharing a level two (L2) cache and all cores sharing a level three (L3) cache. A Dell R900 server rack contains two such Dunnington chips [6] making it twelve cores with three levels of cache topology, L1, L2 and L3. Although these hierarchical cache structures have only recently emerged in commercial CMPs, shared L2 caches have been prevalent for quite some time in dual core and quad core CMPs. In such CMPs, multiple cores sharing an L2 cache leads to a situation where applications running on these cores contend for the shared cache space. This contention can have varying effects on the performance of the simultaneously-executing applications. For instance, an application's performance can be adversely impacted by sharing a cache with another application, whereas the same application can experience minimal adverse impact when running together with some other application. The cache performance of an application is affected by its co-runners that share a cache with it, and further, degree to which the application's cache performance is affected depends not only on its own cache behavior but also on that of its co-runner's. Therefore, co-scheduling threads that have lower contention and hence run well together at the same time is beneficial [50, 51, 52]. This problem gets more complicated in the presence of multi-level cache hierarchies. This is because, different subsets of cores can now have different degrees of cache sharing. For instance, two cores can share both L2 and L3 caches, can share just L2, or share neither. With further increase in the number of cores and multiple caches with deeper hierarchies expected in the future [53], it is vitally important to intelligently map applications to cores in a *cache hierarchy-aware manner* to extract the maximum possible performance.

In this chapter, we address the problem of mapping a workload (a set of single-threaded applications) to the cores of a CMP in the presence of a hierarchical cache structure, and present a mapping algorithm. The presence of different degrees of cache sharing among the subsets of cores introduces different levels of cache contention at different levels of the cache hierarchy. A direct consequence of this contention is the non-triviality of finding an application-to-core mapping which minimizes the overall cache contention effects and improves the overall cache performance.

Our proposed workload mapping scheme starts out by sampling the memory accesses of all applications. The reuse distance distributions are built for all applications in the workload individually using their memory access samples. The performance effects of possible cache contention at different levels of the cache hierarchy are modeled. These reuse distance based models estimate two types of performance effects for each application. The first of these is a measure of the extent to which an application's performance can be adversely affected by other contending applications, and the second measure is the extent to which an application can adversely affect the performance of other (simultaneously-executing) contending applications. We propose a hierarchical grouping technique that uses the reuse-distribution based models to obtain a good application-to-core mapping for a given cache hierarchy and a workload. "Good" in this context means a mapping that reduces the overall cache contention effects (at all cache levels). The grouping algorithm considers all levels of the cache hierarchy progressively and, as a result, the varying degrees of cache sharing among cores are taken into account to reduce the contention effects at all levels.

There have been past studies that analyze the effect of cache contention in the presence of a co-runner [50, 54]. There have also been some online efforts to characterize application behavior [55]. Also, recently, there have been scheduling techniques proposed to address the shared resource contention problem [39] and, algorithms targeted at finding the optimal schedule when the contention between applications is known [56]. *Our work is distinguished from prior efforts in that we take into account multiple levels of the on-chip cache hierarchy, and model in detail the performance effects of applications using reuse distance analysis at different cache levels.* We then use these reuse distance based performance models to group and schedule the target workload on to the cores. Therefore, we propose a complete end-to-end scheme to efficiently map a given workload. It is important to note that, using reuse distance analysis at different levels of the cache hierarchy gives our scheme the ability to identify subsets of cores that have different degrees of sharing and obtain a workload mapping that mitigates potential contention at all levels of the cache hierarchy. To summarize, we make the following contributions:

- In order to motivate the problem, we start out by measuring the performance effects of contention at different levels of a given on-chip cache hierarchy and its effect on overall system throughput.
- We propose *reuse distribution based models* to estimate the cache performance effects of applications due to contention at different cache levels.
- A *cache hierarchy-aware application grouping algorithm* is proposed that tries to find an application-to-core mapping with minimal predicted overall cache-contention-effects.
- We evaluate our proposed mapping scheme on an eight-core and a twelve-core system. In 90% of the cases tested, our scheme computes the best possible mapping, and, the mappings produced by our proposed scheme are within 4% of the best case mappings in all cases. Application-to-core mappings produced by our scheme perform up to 39% better in terms of throughput over a worst-case mapping and up to 30% over the default operating system (OS) based mapping.

4.2 Background and Setup

Hierarchical Caches. Caches and cache hierarchies in CMPs have evolved over the years and span purely private cache organizations, totally shared cache structures and hybrid cache organizations comprising elements of both private and shared cache components. As an example, consider Figure 4.1 which depicts an eight core machine with a three-level hierarchical cache structure. We use this hierarchical cache architecture for the evaluations in this chapter.

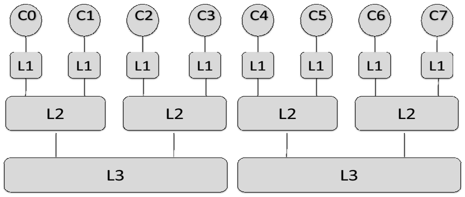
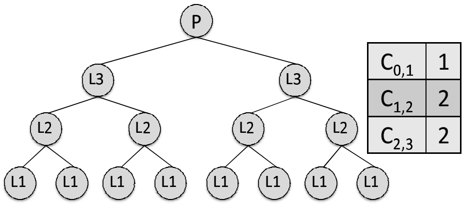


Figure 4.1. A three-level hierarchical cache architecture.

Cache Hierarchy Representation. A hierarchical cache structure can potentially have multiple levels with multiple caches at each level, depending on the underlying topology. Such a hierarchical cache structure can be represented as a tree called the *Cache Hierarchy Tree*. Root of a cache hierarchy tree will be the last level cache if there exists a single last level cache shared by all the cores. If there are more than one last level caches, root of the cache hierarchy tree will be a dummy node, representing the shared off-chip memory. We also define a parameter $C_{i,j}$ to be the number of caches at level i connected to each cache at level j. Therefore, if $C_{i,j} = \delta$, then a total of δ level i caches are connected to each level j cache. Also, level 0 represents the level of the cores. Therefore, a core connected to a private L1 cache is represented by $C_{0,1} = 1$. Figure 4.2 shows the cache hierarchy tree and the $C_{i,j}$ values for the multicore architecture in Figure 4.1.



$C_{0,1}$	1
$C_{1,2}$	2
$C_{2,3}$	2

Figure 4.2. Representation of the three-level cache hierarchy.

Degree of Sharing. In CMP architectures with a hierarchical cache structure, depending on the number of levels of the cache, different subsets of cores can have different *degrees of cache sharing*. Therefore, a hierarchical cache structure creates heterogeneous subsets of cores in terms of cache sharing. The cores present in a CMP can be represented by a set $Cores = \{c_0, c_1, ..., c_m\}$, where m is the number of cores in the system. For every core subset, $S \in Cores$, we define a bit vector called the "*sharing degree*". *Sharing degree* of a core subset S will be $SD(S) = (sd_1, sd_2, ...sd_n)$, where n is the number of levels in the cache hierarchy, and a particular bit sd_j is 1 if all the cores in S share the level j cache, and 0 otherwise. Consequently, there is a bit for each level of the cache which indicates the cache sharing among the subset of cores at that level. Consider the architecture depicted in Figure 4.1. The set of all cores in this architecture can be represented by $Cores = \{c0, c1, ..., c7\}$. We can identify three degrees of cache sharing in this particular topology:

- *High sharing*: Consider the core subset $S \in Cores$. If the cores in S share all levels of the hierarchy except the L1 cache, then S is classified as a *highly sharing*[1] subset of cores. For instance, $S_0 = \{c0, c1\}$ (in Figure 4.1) is a highly sharing subset of cores. The sharing degree vector of core subset S, $SD(S_0)$ is (011). The above subset of cores can experience contention at multiple cache levels (L2 and L3).
- *Medium sharing*: If the cores in the subset $S \in Cores$ share the L3 cache but not the L1 and L2 caches, then the subset, S, is classified as a *medium sharing* subset. An example of this is subset $S_1 = \{c1, c2\}$. In this case, the sharing degree vector is $SD(S_1) = (001)$, and, cores in S_1 experience contention at only the L3 cache level.
- *No sharing*: Core subset $S \in Cores$ is classified as a subset with no sharing if the cores in S do not share any cache. $S2 = \{c3, c4\}$ is an example of this case. Here, the sharing degree vector is $SD(S2) = (000)$. Since no cache is shared, there will be no contention in this case.

The degree of sharing and the corresponding subsets of cores depend on the number of levels in the target cache hierarchy and the way the caches in this hierarchy are connected to each other. For example, in Figure 4.1, there are three different degrees of cache sharing possible, resulting in corresponding contention issues at multiple levels. It is expected that future multicore systems will have deeper cache hierarchies [53], thereby leading to more diverse degrees of cache

---

[1] "Sharing" in this context refers to whether the cores in a subset share a cache or not and has nothing to do with data sharing, which is an application execution characteristic.

sharing.

Experimental Methodology and Setup. All experiments and evaluations presented in this chapter are carried out using Simics [57], which is a full system simulator. Multicore architectures with different number of cores are simulated on Simics. All of the cores simulated in this study are alike and are based on the UltraSparc 3 architecture [3]. The important features of the simulated system are given in Figure 6.1. The cache sizes vary with the different cache architectures we tested. However, in the default architecture (shown in Figure 4.1) used in most of our experiments, the L1 cache is 16KB (4 way associative), each L2 cache (shared by two cores) is 512KB (8 way associative), and each L3 cache (shared by four cores) is 3MB (16 way associative). We also use Simics to obtain the memory access traces of the applications. To map and bind applications to a particular core, we use the Solaris shell command *pbind*. All experiments are performed using applications from the SPEC 2006 benchmark suite [58] with reference inputs. There are a large number of combinations possible while evaluating a workload mapping scheme. For instance, if there are eight applications to be run on eight cores, the number of mappings possible can be as high as 8!, although not all of the mappings are unique as mentioned in Section 4.4. In this chapter, for each application mix, we evaluate all possible mappings, unless otherwise stated. However, due to space constraints, we present results from only a representative sample of the evaluated combinations.

Core architecture	UltraSparc 3+
Core frequency	1 GHz
Operating system	Sun Solaris
L1 cache latency	3 cycles
L2 cache latency	10 cycles
L3 cache latency	40 cycles
Memory latency	260 cycles

Figure 4.3. Major system parameters and their values.

4.3 Motivation

In order to motivate our application-to-core mapping problem, we quantify the differences in performance when different application-to-core mappings are used. Also, we repeat this experiment on architectures with different cache structures.

In other words, we present the differences in performance when different mappings are employed, and study how these differences vary when different cache structures are used. We conduct experiments on three eight-core architectures with different cache structures: a purely shared (all cores share the last level L2 cache), pairwise shared, and the hierarchical three-level cache architecture shown in Figure 4.1. In the pairwise shared architecture, L1s are private and each L2 (last-level) is shared by a pair of cores. Note that the total on-chip cache size (all levels) is the same in all three cases. We used *Perl, Bzip, Gromacs, Sjeng, Lbm, Libq, Gcc* and *Mcf* applications and created workloads with different application-to-core mappings.

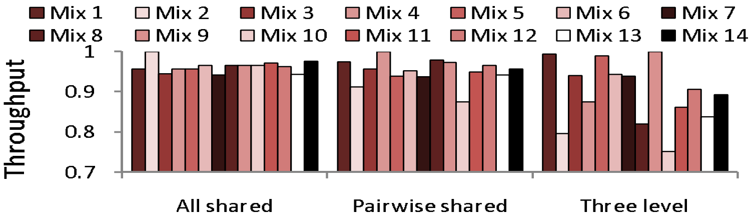


Figure 4.4. Throughput of different application-to-core mappings when executed on a purely shared, pairwise shared and the three-level hierarchical cache architecture. Note here that, throughput is normalized with respect to the one with the highest throughput value in each architecture case.

The overall throughput of the system in terms for all application-to-core mappings on all three of the above mentioned cache hierarchies is plotted in Figure 4.4. The throughput values are normalized to the highest throughput value in each configuration. Firstly, not surprisingly, all mappings result in very similar system throughput on a purely shared cache architecture. This is because there is only one possible degree of sharing among subsets of cores. When a pairwise shared cache hierarchy is used, however, some mappings do much better than the others. More interestingly, when the cache hierarchy shown in Figure 4.1 is used, the mappings which performed very well in the second case do not necessarily perform well now. Also, the variation in performance between different mappings is very high. This is due to the fact that this cache hierarchy has the maximum number of sharing degrees, which is three. This is an important observation since an additional level of shared cache leads to three possible degrees of cache sharing among subsets of cores and, consequently, an additional level of contention, and the combinations of different contention levels result in very high variation among different mappings.

The two takeaway points from this set of experiments are:

- In hierarchical multilevel cache structures, performance difference between the best mapping and the worst mapping can be very high. For example, there is about 30% degradation from the best case to the worst case for the three-level hierarchy in Figure 4.4. Therefore, the importance of finding a good mapping is critical.

- Considering cache sharing at one level is not sufficient. *Degree of sharing*, which indicates cache sharing and contention at multiple levels, needs to be considered when determining a good mapping. For example, a mapping which is good on a pairwise shared cache architecture may not be so good for the architecture in Figure 4.1.

The above motivation concentrated more on the underlying cache structure of the multicore architecture. However, it would also be interesting to know how different applications perform when executed under different degrees of sharing. The question we ask here is whether all applications are affected or affect other contending applications differently. To quantitatively answer this question, we select a single application (*Bzip*) and compare its performance when it is executed under different scenarios. The two scenarios we studied are:

- One companion. *Bzip* runs on core 0 on a pairwise shared cache architecture with another application executing on core 1. We repeat the experiment with different applications on core 1. What runs on the other cores is irrelevant to this experiment (since there is no L3 cache shared across cores 0-3).

- Multiple companions. *Bzip* runs on core 0 of the architecture shown in Figure 4.1 with three other applications on core 1, core 2 and core 3. We repeat the experiment with different combinations of other applications on core1, core2 and core3.

We plot the performance of *Bzip* in each of the above two scenarios in Figure 4.5. As with the system throughput, the performance of *Bzip* depends on not only its immediate companion that shares the L2 cache but also on the other relatively distant companions that share the L3 cache with it. Therefore, any attempt to find a good application-to-core mapping for Bzip should consider the contention effects at *both* L2 and L3 caches (which is indicated by the *degree of sharing*). To check whether other applications behave similarly, we repeated the same experiment with

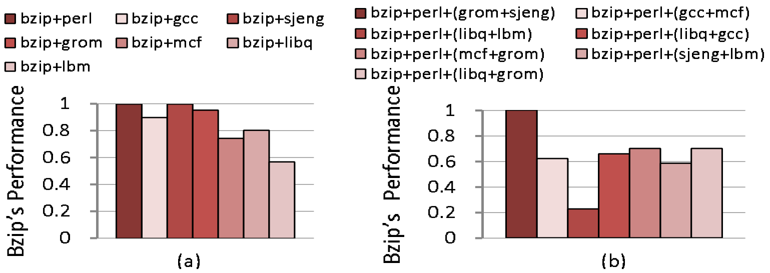


Figure 4.5. (a) shows the performance of Bzip on a pairwise shared cache architecture with different companion applications. (b) shows the performance of Bzip on a three-level hierarchical cache system with different combinations of companion applications. Performance is normalized with respect to the highest performance case.

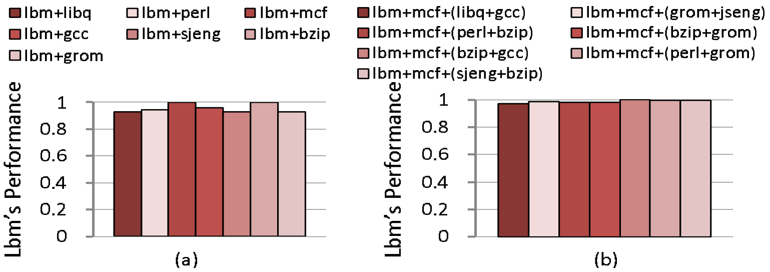


Figure 4.6. (a) shows the performance of Lbm on a pairwise shared cache architecture with different companion applications. (b) shows the performance of Lbm on a three-level hierarchical cache system with different combinations of companion applications. Performance is normalized with respect to the highest performance case.

Lbm and plotted the results in Figure 4.6. As we can observe from this figure, the performance of *Lbm* does not change much with different mappings. We can conclude that the performance of an application and the performance effects of the application on other contending applications depends on the following three factors: (a) *Degree of sharing* of the subset of cores on which the application and other contending applications are executing, (b) Memory access behavior of the application, and (c) Memory access behavior of the contending applications.

Therefore, in order to find a good mapping, we need to consider the following two metrics for each application: (a) Extent to which an application's performance is negatively affected by the other contending applications, and (b) Extent to which an application impacts the performance of the other applications.

4.4 Problem Definition and Roadmap

Computing an application-to-core mapping is akin to finding a good permutation of the given set of applications. If there are M applications, there can be $M!$ permutations (arrangements). It is important to note however that, for a given cache hierarchy, the number of unique application-to-core mappings, P, will be less than $M!$ in most of the cases. This is because not all permutations represent unique mappings with respect to the cache hierarchy. For instance, there is only one unique mapping of four applications to four cores with respect to cache hierarchy in the architecture with four cores and a single L2 cache that all cores share. This is because all subsets of four cores are identical in terms of *degree of cache sharing*. In spite of this, when M is not too small, finding a good mapping by dynamically trying out all possible mappings at runtime is not practical. Therefore, an efficient method to compute a good mapping is required.

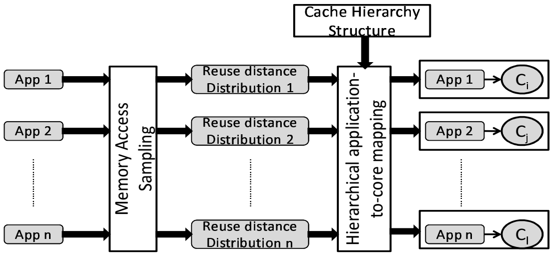

Cache Hierarchy Structure
App 1
App 2
App n
Memory Access Sampling
Reuse distance Distribution 1
Reuse distance Distribution 2
Reuse distance Distribution n
Hierarchical application-to-core mapping
App 1
C_i
App 2
C_j
App n
C_l

Figure 4.7. High level description of our approach to application-to-core mapping.

Figure 4.7 depicts a high-level view of our approach. The first step of our approach is to sample the memory access patterns of each individual application. The second step is to build reuse distance distributions of individual applications. Section 4.5 describes the proposed reuse-distance based modeling of performance effects of applications, when they execute in contention with other applications. Section 6.4 proposes a hierarchical workload mapping scheme. This mapping scheme is implemented as an apriori profiling scheme.The memory accesses of individual applications are sampled, their reuse distance profiles are built, and the workload mapping is determined and applied apriori before the execution begins. Since the sampling and computation are not performed at runtime, the sampling can be performed for longer periods and at multiple points with no runtime overhead.

4.5 Modeling Performance Effects

In this section, we characterize the performance effects of running a given application with other contending applications on a subset of cores with a given degree of sharing. The mentioned performance effects are of two kinds: the first one is the extent to which a given application's performance can be adversely affected by other contending applications, and the second one is the extent to which the given application adversely affects the performance of other contending applications. We define various parameters derived from the reuse distance distribution of an application. These different parameters are essential to characterize the aforementioned performance effects of an application.

4.5.1 Reuse Distance Analysis

Reuse distance is defined as the number of other "unique" cache lines accessed between two contiguous accesses to a particular cache line. A frequency distribution of the reuse distance occurrences is a good indicator of data locality and is called the *reuse distribution* [59]. Figure 4.8 shows a part of the reuse distance distribution of a particular phase of execution of application *Bzip*. Reuse distance is particularly useful since most caches use a variant of the least recently used (LRU) cache replacement policy. In a fully-associative cache with the LRU replacement policy, reuse distance accurately predicts whether an access is a hit or a miss. If the reuse distance is greater than the total number of cache lines in the cache, then the access is a miss; otherwise, the cache access is a hit. Therefore, computing a histogram of reuse distances can accurately predict the miss rate for a fully-associative cache of a given size. This is done by classifying all the frequencies in the reuse distribution histogram with the reuse distance value less than the total number of cache lines as hits and the rest as misses. This reuse distance analysis predicts the cache performance even in the case of associative caches with a small margin of error. Figure 4.8 indicates the hit-miss threshold barrier marked for an L1 cache of size 128 cache lines. All the accesses with reuse distance below the threshold barrier of 128 are estimated to be hits and those with reuse distance higher than 128 are predicted to be misses as shown in Figure 4.8. For instance, using the distribution in Figure 4.8, the cache performance predicted in terms of miss rate is 35.1%, while the actual value on a 4-way set associative cache was 34.3%. Therefore, reuse distance analysis is an accurate way of predicting the cache performance even in the presence of set associativity. Although computing reuse distance distribution can be expensive, there are efficient ways to compute them as discussed by Almasi et al. in [60].

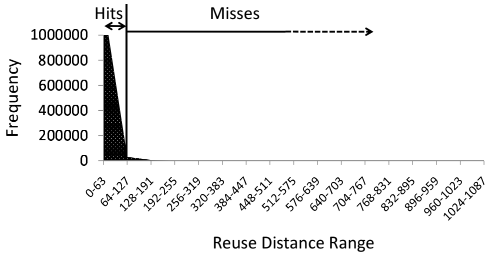


Figure 4.8. A part of the reuse distance distribution of Bzip with the hit-miss threshold barrier marked.

4.5.2 Reuse Distribution Based Parameters

The reuse distance based characterization described in the Section 4.5.1 holds only when the application runs alone with no other applications contending for the cache. More specifically, this characterization is targeted at a scenario where the application runs on a single core processor with no shared caches. However, as described earlier, in the case of multicore architectures with hierarchical cache structures, there will be subsets of cores with different degrees of sharing and hence with different degrees of contention. We now extend this characterization to multicores with hierarchical caches by accounting for these in our modeling. Figure 4.9 shows the reuse distribution of a sample application for illustrative purposes. We define four parameters that can be derived using the reuse distance distribution of an application. These parameters capture the different regions of reuse distance distribution defined by different intervals, are defined for a given level of the target cache hierarchy, and their values will vary with different levels of cache. Let k be the level of the cache considered. Now, consider an application app_a running on core c_i and let Figure 4.9 be its reuse distance distribution. In this plot, $f(R)$ is the frequency of reuse distance R. Let $S \in Cores$ be a subset of cores such that $c_i \in S$ and the kth bit of the "sharing degree vector" of subset S

is set to 1; i.e., $S = \{c_l, c_m...\} \in Cores$, such that $c_i \in S$ and $SD(S)[k] = 1$. Let n be the total number cores sharing the cache at level k, i.e., $n = |S|$. Further, let *totalcache* be the total number of cache lines.

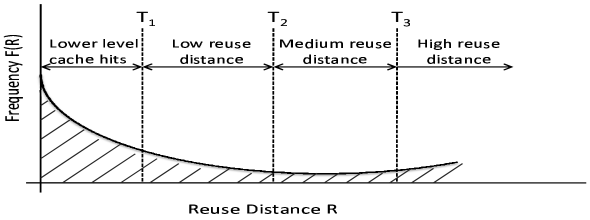


Figure 4.9. Reuse distance distribution of a sample application epoch with different thresholds.

We now define the following metrics for a given cache size *totalcache* and for a given cache level k:

- *Lower level cache hits (LCH).* This parameter estimates the fraction of the reuse distances that will be hits in the lower level of the cache hierarchy and, consequently, will not reach the level k cache. We define "lower level cache hits" as: $LCH_k = \frac{\Sigma_{R=0}^{T_1} f(R)}{\Sigma_{R=0}^{T_\infty} f(R)}$. In this equation, if the lower level cache is the private L1 cache, then the threshold T_1 (shown in Figure 4.9) will be set to the total number of cache lines in the private L1 cache. However, if the lower level cache is a shared cache, our hierarchical application-to-core mapping algorithm sets the value of threshold T_1 as follows: $T_1 = (\frac{1}{n_{low}} + \alpha \times \frac{1}{n_{low}}) \times totalcache_{low}$, where n_{low} is the number of cores sharing the cache at level $k-1$ and $totalcache_{low}$ is the total number of cache lines in the level $k-1$ cache. It is to be noted that, we are determining the threshold T_1 in a conservative manner, i.e., the probability of accesses with reuse distance less than T_1 being a hit in the lower level cache is very high. T_1's value is computed with the insight that shared cache contention is proportional to the number of contending cores. However, the effect of contention can be quite non-uniform at extreme reuse distances, thereby, creating a band of values where T_1 might lie. The tunable parameter α is used to fix the value of T_1 using a very conservative estimation as mentioned above. We later present the chosen value and a sensitivity analysis on α.

- *Low reuse distance (LRD).* We characterize the fraction of the level k cache accesses that have a relatively low reuse distance and therefore have a high possibility of being hits in the level k cache as the "low reuse distance" accesses,

$LRD_k = \frac{\Sigma_{R=T_1}^{T_2} f(R)}{\Sigma_{R=T_1}^{T_\infty} f(R)}$, where T_1 is defined as before and $T_2 = (\frac{1}{n} - \beta \times \frac{1}{n}) \times totalcache$. It is highly likely that the accesses with reuse distance between T_1 and T_2 are going to be hits in the current level (level k). Applications with very high LRD values exhibit very good locality due to a high percentage of low reuse accesses which are likely to be hits even in the presence of high contention. T_1 and T_2 are shown in Figure 4.9. Again, β is a parameter similar to α that helps tune T_2 conservatively based on the effects of contention at reuse distance extremes.

- *Medium reuse distance (MRD).* Medium reuse distance (MRD) parameter estimates the fraction of level k cache accesses that can be either hits or misses depending on other contending applications (i.e., applications running on cores belonging to core subset S). We define MRD as: $MRD_k = \frac{\Sigma_{R=T_2}^{T_3} f(R)}{\Sigma_{R=T_1}^{T_\infty} f(R)}$, where $T_2 = (\frac{1}{n} - \beta \times \frac{1}{n}) \times totalcache$ and $T_3 = (\frac{1}{n} + \alpha \times \frac{1}{n}) \times totalcache$. Note here that, MRD is an estimation of the extent to which an application can be affected by contention from other applications. When contention from other applications running on core subset S is high, these accesses are likely to be misses. On the flipside, when the contention is low, these will likely be cache hits at cache level k. Again, this is a conservative estimate, and T_2 and T_3 are shown in Figure 4.9.

- *High reuse distance (HRD).* When the reuse distance of an access is very high, it is likely going to result in a cache miss. High reuse distance (HRD) parameter estimates the fraction of level k cache accesses that fall under this category. We define HRD as: $HRD_k = \frac{\Sigma_{R=T_3}^{T_\infty} f(R)}{\Sigma_{R=T_1}^{T_\infty} f(R)}$, where $T_3 = (\frac{1}{n} + \alpha \times \frac{1}{n}) \times totalcache$. HRD includes all the accesses which are going to be cold misses and also accesses that have very high reuse distance. When an application runs alone on core c_i with no other contending applications running on other $cores \in S$, then an access with a reuse distance greater than $totalcache$ will very likely be a miss. In the presence of other contending applications, instead of $totalcache$, we use a threshold T_3, which conservatively estimates the fraction of the effective cache space available to this application.

- *Total k level accesses (TotAcc).* This is the total number of accesses to the k^{th} level cache. Therefore, $TotAcc$ for the k^{th} level cache is defined as: $TotAcc_k = \Sigma_{R=T_1}^{T_\infty} f(R)$.

4.5.3 Performance Effects

We now use the above defined parameters to characterize the cache performance of an application in the context of a multicore environment.

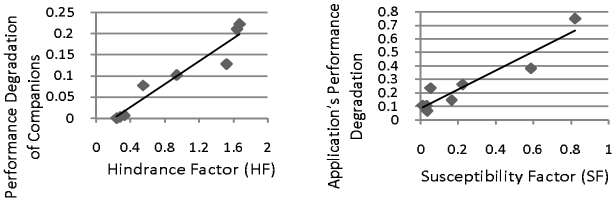
4.5.3.1 Application Characterization

In this approach, we consider the reuse distance distribution of an application in isolation and estimate its performance effects when it executes in a multicore environment with a shared cache hierarchy. To that end, we define two metrics, namely, *Hindrance Factor* and *Susceptability Factor*, for each application. Since both these metrics are defined using the parameters defined in Section 4.5.2, they are for a particular level and size of cache.

- ***Hindrance Factor.*** Hindrance factor of an application estimates the extent to which an application might adversely affect the performance of other contending companion applications. For instance, consider two applications, app_a and app_b, running on two cores with a shared L2 cache. Hindrance factor of application app_a measures the extent to which the cache performance of application app_b is adversely affected due to the contention created by application app_a. We define the hindrance factor of an application (HF) at level k as: $HF_k = HRD_k \times \frac{TotAcc_k}{time}$, where HRD_k and $TotAcc$ are obtained from the reuse distance distribution of an application (as described in Section 4.5.2) and, $time$ is the memory access sampling duration in terms of cycles. If the time of sampling is t sec and frequency of the cores is f Hz, then $time = t \times f$. The hindrance factor measures the number of cache accesses with very high reuse distance per cycle. Therefore, it approximates the number of misses possible per cycle. *HF is actually an estimate of the rate at which an application brings in new data cache lines into the cache.* An application with very high HF value is likely to bring in a large number of cache lines to the cache and, therefore, is likely to occupy more space in the shared cache. Also, such an application can interfere and kick out cache lines that belong to other contending applications. Therefore, a high HF value application is likely to adversely affect other applications due to high contention. Interestingly, an application with very high HF value has relatively lower SF (defined shortly) value and, therefore, is not likely to display good behavior (high performance) in presence of low contention from other applications. However, the actual values of HF and SF are necessary for the performance models, and therefore, we consider both HF and SF metrics.

- ***Susceptability Factor.*** Susceptability factor of an application estimates the extent to which the application's performance can be adversely affected by other contending applications. For instance, as before, consider two applications, app_a and app_b, executing on a pair of cores with a shared L2 cache. Susceptability factor of application app_a measures the extent to which application app_a's performance can be adversely affected by the contending application app_b. More specifically, the susceptability factor of an application (SF) at cache level k is defined as: $SF_k = MRD_k \times \frac{TotAcc_k}{time}$, where MRD_k and $TotAcc$ are obtained from the reuse distance distribution as described in Section 4.5.2, and $time$ is defined as in the case of hindrance factor. Recall that, medium reuse distance (MRD) is defined in Section 4.5.2 as the accesses with reuse distance which is not very high and therefore can be hits if the contention for the cache is low and could turn out to be misses in the presence of high cache contention. Susceptibility factor of an application is a good estimate of how much an application's cache performance can potentially be affected. Therefore, it is important to note that, performance of applications with very high SF values is prone to contention and can easily deteriorate in the presence of high contention. To summarize, our fixed threshold scheme characterizes an application's cache behavior in terms of how much its own performance can be adversely impacted (SF) and how much it can affect the performance of other contending applications (HF).

HF and SF Correlation. We conducted experiments with applications *Perl, Bzip, Gromacs, Sjeng, Gcc, Mcf, Lbm* and *Libquantum* in order to measure the effectiveness of HF and SF in capturing the performance effects. To compute the correlation of HF with the performance degradation of contending applications, we plot the HF values of different applications versus the performance degradation experienced by the contending applications in Figure 4.10(a) (each data point represents an application). The performance degradation is calculated as the percentage degradation from the best case performance and the performance degradation values are averaged over all the applications. We can observe that, the HF metric reflects the trend of performance degradation of the contending applications very well, with a correlation coefficient of 0.96. Figure 4.10(b) plots the SF values of different applications against their own performance degradation due to contending applications. Applications with high SF values suffer a higher performance degradation due to contention than those with lower SF values. The correlation coefficient in this case is 0.94.



(a) HF (a) SF

Figure 4.10. (a) shows the correlation of HF with the performance degradation of the companion applications, and (b) shows the correlation of SF with the application's own performance degradation. The data points represent different applications.

4.6 Reuse Distance Based Workload Mapping

Our application-to-core mapping strategy is carried out in two stages. The first stage, called "*application grouping*", creates groups of applications based on the cache hierarchy tree. The second stage, called the "*group mapping*", maps these groups of applications to subsets of cores available in the target system.

Application Grouping. This stage computes application groups based on the cache hierarchy tree (see Figure 4.2). The groups are formed hierarchically by considering each level of the cache hierarchy tree. "Single Level Grouping" algorithm groups the applications into groups considering a given level of cache. The "Hierarchical Grouping" algorithm goes through all levels of the cache hierarchy, in the process invoking the single-level-grouping algorithm at each level.

Single Level Grouping. Single-level-grouping algorithm shown in Figure 4.11 takes an input set, IS, of *elements* and groups the elements into p groups. p is the total number of caches at the cache level k. In other words, p is the number of nodes at level k of the cache hierarchy tree T. The elements in the input set, IS, can be either applications or groups of applications. This is because, before the hierarchical group algorithm invokes the single-level-grouping at some level j, it would have invoked the single-level-grouping at the previous level $j-1$, which

```
$Single\_Level\_Grouping$
Inputs: $k$ - cache level; $T$ - cache hierarchy tree
    $IS$ - set of elements to be divided into groups
    $mem\_limit$ - threshold to characterize cache behavior
Output: $groups$ - set of groups computed
        //$groups$ is set of subsets of $IS$, such that,
        $g_i, g_j \in groups$, $g_i \cap g_j = \emptyset$
        and, $g_0 \cup g_1 \cup \ldots g_n = IS$, where, $n = |groups|$
Initialization:
    $m$ - $|IS|$;  $num\_nodes$ = number of nodes in $T$ at level $k$
    $group\_size = \frac{m}{num\_nodes}$; $groups = \{\}$
Grouping:
    while ($IS \neq \emptyset$) //until all elements are grouped
        $curr\_group = \{\}$
        if $\exists elem_x \in IS$, such that,
        $HF_k(elem_x) > mem\_limit$ // $HF_k$- hindrance factor
            $first\_elem = elem_x$
        else
            select $elem_y \in IS$, such that $SF_k(elem_x)$ is max
            where, $SF_k$ is the susceptability factor.
            $first\_elem = elem_y$
        $curr\_group = \{first\_elem\}$,$IS = IS - first\_elem$
        for $num \leftarrow 1$ to $group\_size$:
          if ($HF_k(first\_elem) > mem\_limit$)
              if $\exists elem_j \in IS$,
              such that, $HF_k(elem_j) > mem\_limit$
                $curr\_group = curr\_group + elem_j$,
                $IS = IS - elem_j$
              else:
                select $elem_k \in IS$, such that, $SF_k(elem_k)$ is min
                $curr\_group = curr\_group + elem_k$,
                $IS = IS - elem_k$
          else
              select $elem_j \in IS$ with minimum $SF_k(elem_j)$
              $curr\_group = curr\_group + elem_j$,
              $IS = IS - elem_j$
        end for
        $groups = groups + curr\_group$
    end while
    return $groups$
```

Figure 4.11. Single-level grouping algorithm using *fixed thresholds.*

would have created groups at level $j-1$. Therefore, at level j, the set of these groups are further clustered into larger groups. We use a heuristic that prunes the search space to a very small space. Our algorithm uses *Hindrance Factor* and *Susceptibility Factor* of applications derived from the reuse distance based parameters (see Section 4.5.3) to make grouping decisions. Note that, HF and SF values for the k^{th} level cache are used here (i.e., HF_k and SF_k). This is important because different invocations of the algorithm for different cache levels use different HF and SF values of applications.

The goal of our single level grouping algorithm is to group applications such that the adverse effects of cache contention are mitigated and performance is improved at runtime compared to other possible groupings. The size of the groups to be formed is determined by considering the total number of caches at level k and computing the number of applications per level k cache assuming an equal division of applications.[2] The algorithm starts out by choosing and grouping elements with very high HF value together. For an application, app_a, its HF value, $HF(app_a)$ is classified as "very high" if $HF(app_a) > mem\_limit$. Here, mem_limit is a threshold value chosen such that, applications with $HF > mem\_limit$ adversely affects the performance of the contending applications, especially if the contending applications have a high SF (*susceptibility factor*). This however is a tunable parameter. The reasoning behind grouping all these "badly behaving" applications together is twofold. Firstly, by quarantining these applications together in a different group, other applications with high SF value and low HF value are not adversely impacted by these applications. The second reason is that these applications have such high HF values that their cache performance will be poor even if they are running alone. In other words, even in case of lesser or no contention, these applications achieve very little performance gain compared to the applications with high SF and low HF values. Therefore, it may make sense to group and schedule them together. Note that, the algorithm may run out of other similar high HF value applications. This is because of the fixed size of groups as mentioned before. In that case, the high HF value application is grouped with an application with the minimum SF value. This is done with the intention that the high HF value application with high contention impacts a low SF value application lesser than a high SF value application. After all the high HF values are grouped, our algorithm starts grouping applications with relatively low HF values. The strategy employed by the algorithm to group these applications is different from that of the very high HF applications. The algorithm tries to group an application with high SF with another application with a low SF value, because a high SF value means that an application has a lot of accesses which can potentially be hits under low contention but can be misses under high contention environment. Therefore, the high SF value application has a higher number of cache hits due to low contention

---

[2] "Size" in this context refers to the number of applications in a group.

from the other application since it has a lower SF value. Also, the other application with lower SF value anyway has very few accesses, which can be affected by contention and hence is not affected too much by the high SF application. Computation and usage of HF and SF values are a key step in the above algorithm. As we mentioned before, an element of the input set (IS) can be an application or a group of applications. The HF and SF values have an additive property in our algorithm, that is, the HF and SF values of a group of elements will be equal to the sum of the HF and SF values of the elements present in the group.

Hierarchical Grouping. The previous section described the grouping of applications in a cache-sharing aware manner but considering cache sharing at a given single level in the cache hierarchy. Figure 4.12 describes our approach that computes application groupings based on all levels of the cache hierarchy. This algorithm hierarchically groups applications by calling single level grouping algorithm (Figure 4.11) at each level of the cache hierarchy tree. This grouping scheme starts from the first shared level of cache and hierarchically groups the applications at each level until the root of the hierarchy tree, T, is reached.

```
$Hierarchical\_Grouping$
 Inputs: $AS = \{app_0, app_1, .., app_m\}$, $T$ - cache hierarchy tree
 Output: $hgroups$ - set of groups after hierarchical grouping
          //each $hg_i \in hgroups$ can be a set of groups
 Initialization:
     $level = k$, such that, $C_{k-1,k} > 1$ and $C_{k-2,k-1} = 1$
     // $level$ is the lowest shared level in $T$
     $top$ = root level of the cache hierarchy tree, $m = |AS|$
Hierarchical Grouping:
     $hgroups_{old} = IS$; $hgroups_{new} = \emptyset$
     while ($level < top$)
        $hgroups_{new} =$
        $Single\_Level\_Grouping(level, hgroups_{old}, T)$
          $hgroups_{old} = hgroups_{new}$; $level = parent(level)$
          //update $level$ to the parent of the current level in the $T$
     end while; return $hgroups$
```

Figure 4.12. Hierarchical application grouping algorithm.

Mapping Groups to Cores. Once the hierarchical application groups have been created, the applications in these groups are mapped to cores based on their groups and the sharing degree of the subsets of cores. This mapping algorithm takes the grouping determined by the hierarchical grouping algorithm as input and starts assigning these groups to cache nodes at each level of the hierarchy.

The mapping begins with the root node of the cache hierarchy tree and proceeds downwards till the leaf node level, at which point the determined grouping is assigned to the cores.

Illustration. Figure 4.13 shows an illustration of how our mapping approach works in practice. Our default CMP architecture is considered here (see Figure 4.1), and we have eight applications. The hierarchical grouping algorithm starts out at the bottom level (leaf node level) and moves to towards the root, calling the single-level-grouping algorithm at each level on the way. The grouping returned by our single-level-grouping algorithm at each level is shown in Figure 4.13. The mapping algorithm takes the generated grouping as input and traverses from the root to the leaf node level of the tree, assigning groups to cache nodes at each level. When the private cache level is reached, the assignment is complete.

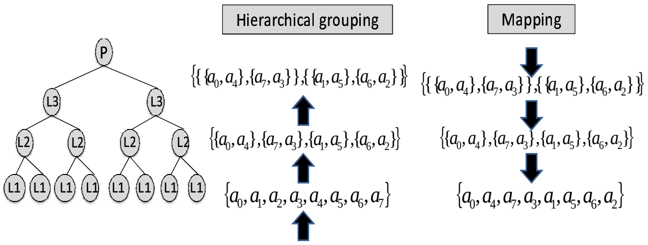

Hierarchical grouping
Mapping
P
L3
L3
L2
L2
L2
L2
L1
L1
L1
L1
L1
L1
L1
L1
$\{\{\{a_0,a_4\},\{a_7,a_3\}\},\{\{a_1,a_5\},\{a_6,a_2\}\}\}$
$\{\{a_0,a_4\},\{a_7,a_3\},\{a_1,a_5\},\{a_6,a_2\}\}$
$\{a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7\}$
$\{\{\{a_0,a_4\},\{a_7,a_3\}\},\{\{a_1,a_5\},\{a_6,a_2\}\}\}$
$\{\{a_0,a_4\},\{a_7,a_3\},\{a_1,a_5\},\{a_6,a_2\}\}$
$\{a_0,a_4,a_7,a_3,a_1,a_5,a_6,a_2\}$

Figure 4.13. Illustration of the grouping and mapping steps.

4.7 Experimental Evaluation

The experimental setup and the methodology described in Section 4.2 are used in all of the experiments. In order to build the reuse distance profiles, the applications were sampled for 100 million instructions. After preliminary experiments, we selected the values of α and β parameters (mentioned in Section 4.5.2) to be 0.3 and 0.2, respectively, and set the value of *mem_limit* (mentioned in Section 6.4) to 1.

Average Results. We evaluated the performance of our proposed hierarchical mapping scheme using 12 randomly selected workloads built using applications from the SPEC 2006 benchmark suite [58]. Figure 4.14 presents the throughput comparison of our proposed scheme over the best case mapping, worst case mapping and three different runs of the default OS scheduling scheme for four representative workloads. In this chapter, by throughput, we always refer to IPC throughput unless otherwise mentioned. Figure 4.14 also plots the average throughput comparison (averaged over all 12 workloads). The workloads considered here are $Work1$= {*Sphinx, Milc, Libquantum, Lbm, Gobmk, Hmmer, Bzip, Perl*}, $Work2$= {*Gromacs, H264, Hmmer, Lbm, Mcf, Sphinx, Gobmk, Perl*}, $Work3$= {*Sjeng, Gobmk, Gcc, Mcf, Lbm, Libquantum, Perl, Bzip*} and $Work4$= {*Hmmer, Sphinx, Sjeng, Perl, Lbm, Libquantum, Bzip, Gobmk*}.

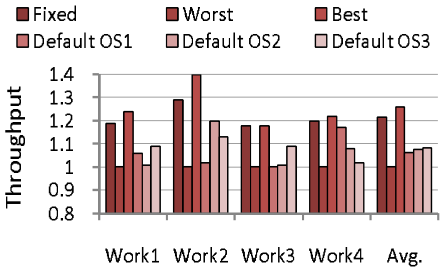


Figure 4.14. Throughput comparison for four representative workloads and the average case (over all 12 workloads) on the eight-core CMP.

The application grouping scheme finds the best mapping in the case of $Work3$ and $Work4$, while it finds a mapping that performs slightly worse than the best mapping for $Work1$ and $Work2$. In most cases, our proposed scheme finds the best possible mapping, while in the case of the other sets, mapping computed by our proposed scheme is within 4% of the best case mapping. Also, mappings computed by our scheme are up to about 40% (average of 25%) better than the worst case mapping in terms of system throughput. In the above experiment, the best case and worst case mappings are determined by trying out all possible combinations of mappings. For the default OS based scheme, we run the applications on the eight-core system and do not bind the applications to cores. Interestingly, since the OS does not consider the cache hierarchy and schedules applications randomly (with respect to cache hierarchy awareness), different runs of the OS based scheme yield different mappings and hence, different results. In order to demonstrate this, we run the default OS based scheme multiple times.

Workload Instance. In order to analyze the performance impact of our proposed scheme on both the workload throughput and the individual application performance, we consider a single workload comprised of *Perl, Bzip, Gromacs, Sjeng, Gcc, Mcf, Lbm* and *Libquantum* applications. Figure 4.15 shows the throughput achieved by the application grouping scheme alongside the best case mapping, the worst case mapping, and the default OS based mapping. One can observe from these results that, the application grouping scheme achieves performance benefits of about 32% over the worst case mapping and up to 30% over the default OS mapping.

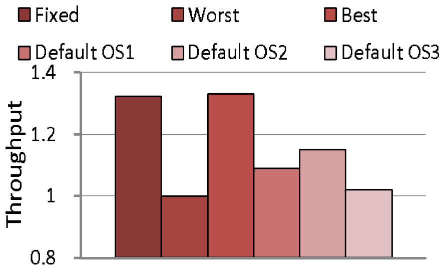


Figure 4.15. Throughput comparison on an eight-core CMP when workload of perl, bzip, gromacs, sjeng, gcc, mcf, lbm and libq applications is executed.

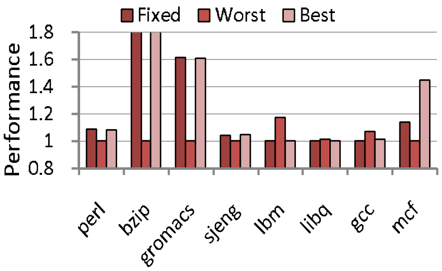


Figure 4.16. Performance comparison of applications on the eight-core CMP when a workload of perl, bzip, gromacs, sjeng, gcc, mcf, lbm and libq applications is executed.

In Figure 4.16, we show the performance of the individual applications under different schemes. Note that, we do not show the individual application performances in the OS based scheme. This is because, since in the OS based scheme we do not bind the applications to cores, it is hard to determine which application runs on which core. One can see from Figure 4.16 that, *Bzip* and *Gromacs* can significantly improve their performance when a good application-to-core mapping is employed. This is because they have a very high *SF* values. *Perl* and *Sjeng* can also perform better when the mapping is good. The application grouping scheme finds the best possible mapping in this case.

12 Core System. We also conducted experiments on a twelve core CMP, where each pair of cores share an L2 cache and each group of six cores share an L3 cache. Each L2 cache is 512 KB (8 way associative) and each L3 cache (shared by 6 cores) is 6 MB in size. The twelve applications run in this experiment are *Perl, Bzip, Gromacs, Gobmk, Sjeng, Hmmer, Sphinx, Gcc, Mcf, Milc, Libquantum* and *Lbm.* Figure 4.17 and Figure 4.18 present, respectively, the throughput and application performance results in this case. Our scheme outperforms the worst case mapping by about 17% in terms of the overall system throughput.

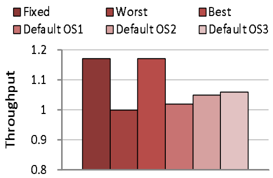


Figure 4.17. Throughput comparison on a 12-core CMP.

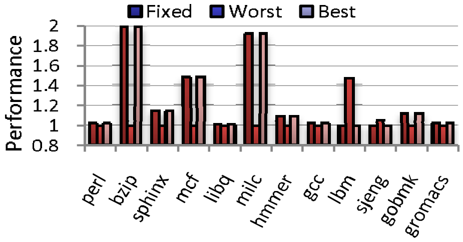


Figure 4.18. Performance comparison of applications on a 12-core CMP.

Sensitivity Analysis. We repeated the eight-core experiments mentioned above with larger on-chip caches (1MB L2s and 4MB L3s). Our scheme outperformed the worst case mapping by about 17% and the default OS mappings by around 7%. Finally, we also evaluated our scheme with increased sampling lengths, which resulted in the same mappings as before, and therefore, the same performance. We also experimented with different α and β values, and found that the values of 0.3 and 0.2, respectively, performed the best across all workloads. Also, changing the values slightly in either direction yielded very little performance difference in this set of workloads. Therefore, the behavior and the performance of our proposed scheme is almost independent of the α and β values employed.

4.8 Discussion of Related Work

Cache management techniques use architectural level modifications [61, 62] to improve shared cache performance. In comparison, cache partitioning techniques explicitly partition the shared cache [63, 64, 17, 19] for performance isolation. There have been research efforts aimed at predicting cache contention in terms of performance degradation due to cache contention [50, 65]. Song et al. model the L2 cache behavior for a set of scientific applications on CMPs [54]. Federova et al. propose an L2 cache aware scheduling algorithm based on metrics such as missrate [66]. Xie et al. aim to make a broad characterization of programs at runtime using metrics such as miss rate [55]. Gang scheduling of threads of parallel jobs concurrently provides performance benefits as proposed by Jette et al [67]. Bulpin et al. propose to use hardware performance counters to bind threads to processors. [51] and [52] aim to find a symbiotic job schedule which runs well together after trying out different combinations. DeVuyst and Tullsen propose an unbalanced scheduling scheme that yields power and performance benefits [68]. Federova et al. propose a scheduling algorithm to improve performance isolation [69]. [70] proposes OS enhancements that use hardware monitors to improve the capabilities of OS to manage CMP resources. In [56], performance degradation among different combinations of applications is estimated and a co-scheduling scheme is proposed using these estimated degradation values. Tam et al. propose to cluster threads based on the data sharing between them [71] when the threads of a single application can share data. Chen et al. also propose to schedule threads for constructive cache sharing [72]. In [73], Zhang et al argue that when threads belong to the same application, alternate schedules do not enhance cache sharing. Reuse distance analysis has been been studied extensively in the context of single, sequential execution [60, 59, 74]. In [74], Beyls et al. show that reuse distance analysis of a sequential execution can reflect its cache performance in terms of miss rate very accurately even in the presence of associativity. There have also been proposals to efficiently calculate reuse distance distributions [60]. In [75], authors try to maintain multiple reuse stacks in the case of CMPs to gather reuse distances. Some of the prior works mentioned above try to measure cache contention when there are two cores sharing a cache [50], while others try to compute contention aware schedules [39]. There has been *no* prior work however to model the performance effects in detail when there are *multiple levels in the cache hierarchy* and multiple associated cores, where different subsets of cores can have different degrees of sharing. There have been efforts to find near-optimal schedules which assume the knowledge of performance degradation between different combinations of applications [56, 76]. The problem with such schemes is that, not only is computing such performance degradation values difficult but also the fact that these performance degradation values depend on the cache structure. For a given architecture with a particular hierarchical cache-structure, our work models in detail the effect of cache contention and cache interference at different levels of the cache hierarchy using the reuse distance profiles of the individual applications. Our scheme then uses these models to compute a smart workload mapping. The usage of reuse distance analysis at multiple levels of the cache gives us the ability to consider all different degrees of sharing between subsets of cores before making the workload mapping decisions.

4.9 Conclusion and Future Work

In this chapter, we showed that the reuse distance analysis is very effective in predicting the performance effects of an application when it is executed with other contending applications. Based on our reuse distance based performance modeling, we then studied a workload mapping strategy targeting CMPs with hierarchical caches. This strategy is very effective in choosing a good mapping with performance benefits of up to 39% over the worst-case mapping and up to 30% over the default OS based mapping. As part of our future work, we intend to study a runtime scheme that can perform our workload mapping scheme dynamically during execution.

Chapter 5

Intra-Application Cache Partitioning

5.1 Introduction

In current multicore systems, a shared L2 cache has become the dominant on-chip cache alternative, thanks to its efficiency in utilization and flexibility in dynamic allocation [3, 2]. However, as discussed in Chapter 1, inter-thread contention is a major issue in such shared caches. This contention for the shared cache from different threads (executing on different cores) can have varying impact on those threads. In such a scenario, performance of one or more of the contending threads can be adversely affected [17, 72]. The commonly used LRU replacement policy can lead to threads with not so good cache behavior occupying most of the shared cache with very little performance gain, while other threads with possibly good cache behavior starve [77, 78]. Therefore, efficient and smart management of an important shared resource such as L2 cache is vital.

In this chapter, we study the cache space partitioning problem when the contending threads belong to the same application. That is, as opposed to existing cache partitioning schemes, we investigate *intra-application* cache partitioning. Most parallel applications targeting shared memory systems are programmed to contain one or more parallel sections, which are bound by synchronization constructs such as barriers. Performance of a parallel section is always determined by the slowest thread, also called the *critical path thread*. A thread with excellent cache behavior does little to speed up the application performance if the other application threads have really poor cache behavior. When large multithreaded applications are executed on multicore machines, such differences in thread behavior can severely limit the performance benefits of utilizing large number of cores. Efficacy of previously described cache space partitioning and management techniques is questionable in this scenario, where the contending threads belong to the same application. This is because, those techniques are agnostic to inter-thread relationships and, consequently, try to either optimize throughput or maintain cache fairness. However, as we mentioned above, performance of a multithreaded application is largely determined by the performance of the critical path thread and therefore most of the prior techniques prove ineffective in improving the application performance. We claim that, when partitioning the shared cache among the threads of the same application, the goal should be neither throughput improvement nor fairness, but in fact speeding up the critical path thread.

An important point to consider when contending threads belong to the same application is the net effect of such cache contention. For starters, threads from the same application can share data, and consequently, effects of contention need not always be adverse. When threads share data, a simple shared cache without any sort of partitioning may perform well in certain cases, due to possible constructive sharing. In fact, absence of cache data sharing in private caches appears to be a major drawback for single multithreaded application execution on such systems. On the flip-side, in shared cache architectures, inter-thread cache evictions (one thread evicting the data brought in by a different thread) happens to be a major performance concern for most parallel multithreaded applications.

We propose a dynamic, runtime system based *cache partitioning* scheme to partition a shared cache among threads of a single application to enhance the overall performance of the multithreaded application. We propose to dynamically determine the cache requirements of individual threads based on their cache performances and partition the cache space such that the slowest thread gets most of the cache space. We discuss two such dynamic, run-time system based cache partitioning techniques. The first technique partitions the cache space at each execution interval based on the cycles-per-instruction (CPI) values of the individual threads, such that the slowest thread (critical path thread), which is the thread with the highest CPI, gets a larger portion of the cache. The second technique dynamically builds a cache model through curve fitting for each thread and finds a cache partition that minimizes the slack time at each interval. In this context, slack time is defined as the difference between thread speeds. The goal of this algorithm is to try to ensure that the threads progress at approximately same speeds by allocating lesser cache to threads with good cache hit rates and more cache to threads with low cache hit rates.

We observe that our technique achieves application speedups up to 15% over an unpartitioned shared cache, up to 23% over a statically partitioned cache, and up to 20% over a prior throughput-oriented scheme.

5.2 Background and Setup

5.2.1 Architecture Specification

In this work, we consider a CMP with a shared level 2 (L2) cache. Our target system, unless otherwise mentioned, is a four core CMP, with an L2 cache shared by all cores. This L2 cache is assumed to be highly associative. Each of the cores also maintains private L1 data and instruction caches. We want to make it clear that, whenever we talk about cache partitioning in this work, we always refer to cache way partitioning [64]. Therefore, assigning more cache resources to a thread is synonymous to assigning more cache ways to the thread in our context.

5.2.2 Parallel Program Structure

Most multithreaded shared-memory parallel programs generally have several parallel code sections interspersed with sequential sections. Sequential sections might perform tasks such as data initialization and data collation. Also, synchronization might be needed between any two sections to maintain data integrity and avoid race conditions. For such synchronization purposes, these parallel sections employ constructs such as *barriers*. Figure 5.1 shows the structure of a typical parallel program (on the left) and also depicts the execution progress of a sample parallel section (on the right), assuming four threads. Execution proceeds from a parallel section to the next stage (possibly sequential) only after all threads in the parallel section complete, and consequently, reach the barrier. The execution time of such a parallel section is determined by the time taken for the slowest thread to reach the barrier, also termed as the *critical path thread*.

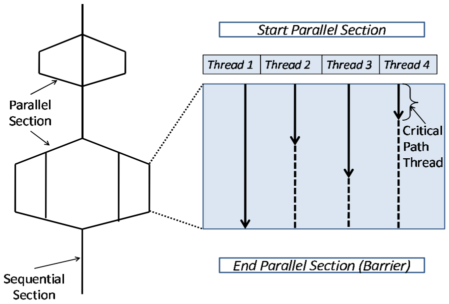


Figure 5.1. Left: A sample execution for shared-memory multithreaded application. Right: Progress of threads in a parallel section at a particular point during execution.

5.3 Motivation

In this section, we present the motivation for our intra-application dynamic cache partitioning scheme.

5.3.1 Why Intra-Application Cache Partitioning?

Runtime behavior of parallel multithreaded applications exhibit certain characteristics that make them amenable to intra-application cache partitioning. In this section, we identify those characteristics and describe how each of those characteristics motivates the need for intra-application cache partitioning.

5.3.1.1 Performance Variability

In Section 5.2.2, we discussed the structure of a typical parallel program and the possible variability in execution speeds of the different threads. In this section, we analyze with empirical support, if this is indeed the case. Figure 5.2 shows the overall performance [1] of each of the four threads of nine parallel benchmarks over 50 execution intervals. The performance of the application threads are *normalized* to the fastest thread in Figure 5.2. As we can see, the applications exhibit

---

[1] We consider the inverse of execution time as performance.

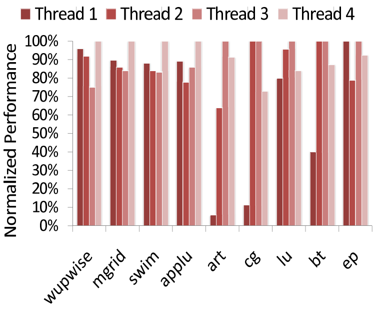


Figure 5.2. Performance of individual threads of the application normalized to the fastest thread.

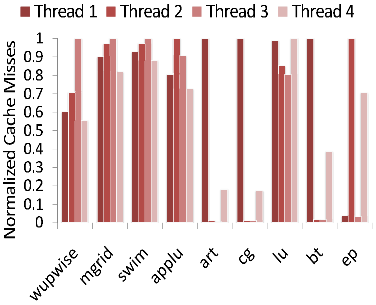


Figure 5.3. Number of L2 misses incurred by each individual thread normalized to the thread with highest misses.

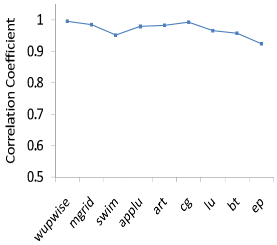
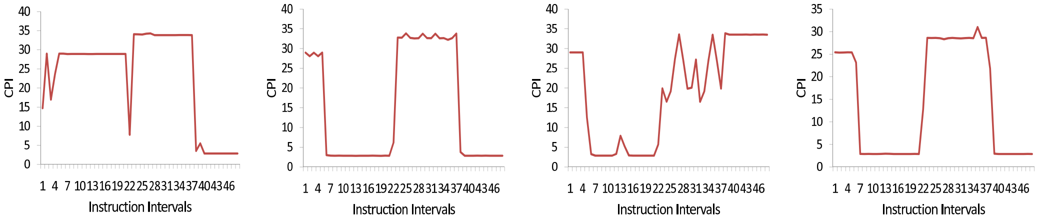


Figure 5.4. Correlation coefficient between the number of L2 cache misses and the corresponding CPI values.

a wide variability in the performance of the threads. More importantly, in every application, the critical path thread (represented by the lowest bar) is considerably slower than the other threads and hence determines the performance of the overall application. In general, variation in performance (slack time) among the application threads is very high. For instance, in MGRID, although thread 3 performs exceedingly well with a CPI of 7.1, the application performance is held back by thread 2, which has a comparably poor CPI value of 11.5.

In order to study this variability among the application threads, we collected cache performance statistics for all the above applications during their runs. Figure 5.3 plots the cache performance of the four application threads in terms of the L2 misses incurred by each of the threads. The values are *normalized* to the thread with the highest number of L2 misses. As we can clearly see from Figures 5.2 and 5.3, the variability in the overall performance of these threads correlates very closely with the variability in their cache performances. More specifically, if the performance of a thread is low in Figure 5.2, its cache miss count is high in Figure 5.3, and vice versa. This correlation between the CPI values and the number of cache misses can be clearly seen in Figure 5.4. This figure plots the correlation coefficient between the number of L2 cache misses and the CPI value for the applications in our experimental suite. We can infer, from this plot, a strong linear dependence between number of L2 cache misses and CPI for these applications, with an average correlation coefficient value of 0.97.

Further, we plot the performance of the individual threads of SWIM application



(a) Thread 1. (b) Thread 2. (c) Thread 3. (d) Thread 4.

Figure 5.5. CPI values of the four SWIM application threads during 50 consecutive execution intervals.

over 50 contiguous execution intervals in Figure 5.5. It can be observed from this graph that, in addition to the variability across the performances of individual threads, there is also a variation across execution intervals.

This is due to the fact that a multithreaded application, typically goes through different phases during its execution [79]. There can be various reasons for this phase behavior. During one stage, a thread can be memory bound with poor cache performance; during another, it can have really good cache performance; some parts may not be memory intensive at all; some other parts can have poor branch predictor performance [79]. In order to identify the main factor, we plot the L2 cache misses corresponding to thread 2 in Figure 5.5(b) during the same 50 contiguous execution intervals in Figure 5.6. As before, we observe a clear correlation between the CPI across time and the corresponding cache misses during the same time interval. Therefore, due to this variability across time, the critical path thread may change from one execution phase to another.

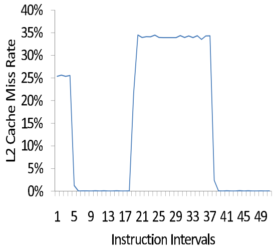


Figure 5.6. L2 misses during 50 execution intervals of thread 2 of SWIM benchmark.

To summarize, different threads belonging to the same application have very different cache requirements from one another and further, these cache requirements also vary over time.

5.3.1.2 Cache Interaction Across Threads

Application threads executing on different cores sharing a cache can interact in different ways. In this section, we describe and quantify both the amount and the kind of cache interactions exhibited by the application threads. Firstly, we ran our nine applications on the target CMP and collected the inter-thread cache interaction statistics. By *inter-thread cache interaction*, we mean the percentage of cache accesses that are inter-thread accesses. In this context, we specify a cache access to be an inter-thread cache interaction if a previous access to the same cache line was from a different thread. On the other hand, if two contiguous accesses to a cache line are from the same thread, then it is an intra-thread interaction. Recall that we use the terms "thread" and "core" interchangeably since we consider a single thread executing on each core. Another important point to note here is that cache interaction covers all accesses and not just misses. Figure 5.7 shows the contribution of inter-thread cache interactions (when considering all interactions). As we can clearly see, there is a considerable amount of inter-thread interaction in these multithreaded applications, averaging about 11.5% of all cache interactions.

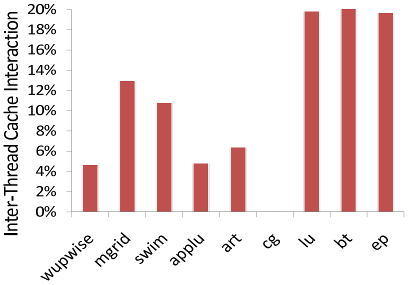


Figure 5.7. Percentage of cache interaction that happens to be inter-thread.

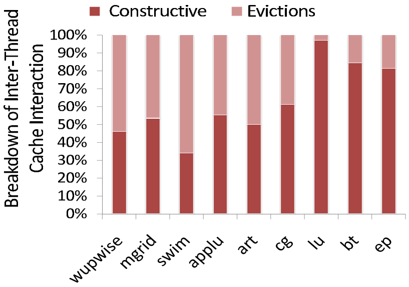
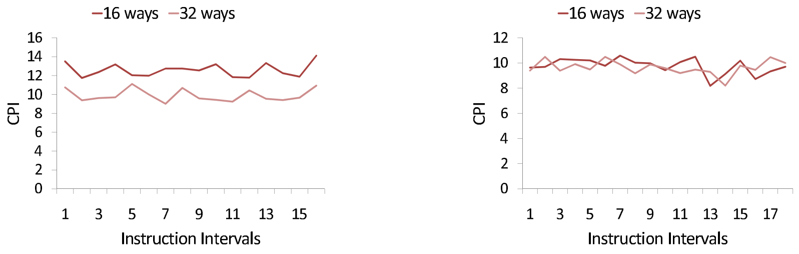


Figure 5.8. Percentage of constructive inter-thread cache interactions.

We also studied the nature of these inter-thread interactions. We describe *constructive* inter-thread cache interaction to be the percentage of inter-thread interaction that happen to be cache hits. Therefore, a set of two contiguous accesses to a cache line is considered to be constructive interaction if the second access is a hit. To rephrase, constructive interaction happens when a data element brought into the cache by a thread is also used by another thread before it gets displaced, thereby helping the latter thread to improve its performance. It is to be noted that, constructive inter-thread cache interaction is a result of data sharing between the interacting threads. We plot the breakdown of constructive and destructive (evictions) inter-thread interactions in Figure 5.8. We can infer from this graph that, not all inter-thread interactions are constructive. A significant amount of destructive inter-thread interactions in the form of evictions can be seen in Figure 5.8. In such a partitioned shared cache, a thread can access a cache line present in a cache partition belonging to another thread, thereby, enabling constructive inter-thread cache sharing. However, a thread cannot evict a cache line belonging to another thread's cache partition, thereby, preventing destructive inter-thread cache interference. In other words, cache space partition is in terms of eviction control.

5.3.1.3 Cache Sensitivity Variability

Another facet of a parallel program's behavior is the cache sensitivity of individual threads of the program. We ran a four-threaded SWIM application multiple times, but each time using different cache sizes. We increased the cache size from 32 KB progressively until 1MB. We want to reiterate that, to increase cache size, we simply add more ways. Therefore, the increase in the total cache size is accompanied by a corresponding increase in associativity. For instance, a 32KB cache is 2 way associative, but a larger 64KB cache will be 4 way associative. We show the CPI for two of the threads of this application for 16 and 32 ways in Figure 5.9.



(a) Thread 1 with 16 and 32 ways.

(b) Thread 2 with 16 and 32 ways.

Figure 5.9. CPI curves for two threads of SWIM when executed with 16 and 32 ways. Clearly, thread 1 shows considerably more improvement when the number of ways is increased from 16 to 32, when compared to very little improvement exhibited by thread 2

An important observation that can be made is that, the individual threads have variable sensitivity to cache capacity. While a cache size increase may benefit one thread, it might not improve another thread's performance. In Figure 5.9, when the cache size is increased from 16 to 32 ways, thread 1 exhibits a higher CPI reduction, as compared to almost no CPI reduction in the case of thread 2. Therefore, thread 1 is more sensitive to cache than thread 2 since thread 1 benefits a lot more by cache size increase than thread 2. This heterogeneity in cache sensitivity among threads of the same application is an important observation because taking cache resources away from a cache insensitive thread may not be detrimental, as far as overall performance is considered. Therefore, cache resources can possibly be taken away from a cache insensitive thread without much affect on its performance. On the flip side, if the critical path thread is cache insensitive, then giving more cache ways to it might not be too beneficial in practice. Therefore, cache sensitivities of threads dictates how effective a cache partitioning scheme can be.

5.4 Dynamic Cache Partitioning

Our dynamic cache partitioning scheme is applied at the granularity of *execution intervals* of around 15 million instructions. That is, at the end of each interval, we optimize for the next interval[2]. The dynamic cache partitioning scheme gathers execution counter values such as cache hits/misses, cycle counts and instruction counts for each thread during each execution interval. At the end of each such interval, the cache space is partitioned based on individual thread performances, so as to speed up the critical path thread, at the cost of other threads.

5.4.1 CPI Based Partitioning

CPI based cache partitioning is a scheme which collects execution characteristics during each interval and partitions the cache space based on the thread cycles-per-instruction (CPI) values [12]. The thread with a high CPI receives a larger portion of the cache and those with lower CPIs receive lesser portions accordingly. This is done with a hope that a thread with a high CPI (critical path thread) can

---

[2] Note that an (execution) interval can contain multiple parallel sections, and similarly, a parallel section can span multiple execution intervals.

improve its performance with a larger cache share, thereby, improving the overall application performance. In this scheme, we start out with equal cache partitions during the first interval. At the end of each interval, the execution characteristics are collected and the CPI values for each of the threads are computed. The cache space is then partitioned based on these computed CPI values. The formulation used to decide the cache partitions is:

$$partition_t = \frac{CPI_t}{\sum CPI_i} \times Total\_Cache\_Ways$$

That is, the number of cache ways assigned to each thread is proportional to the CPI value of the thread.

5.4.2 Dynamic Model Based Partitioning

The main drawback of the previous scheme is its naivete in assuming a particular cache sensitivity metric. It lacks the knowledge of how the CPI value of the thread may change when a cache way is given to it or when a cache way is taken away from it. Therefore, a simple CPI based cache partitioning may not do very well, and in fact, may harm performance in certain cases. Therefore, we propose a *dynamic learning based* algorithm that considers not just the thread CPIs during the current interval but also the individual thread CPI curves so that the cache space can be partitioned more accurately. The core of this scheme is a *runtime thread performance modeling*, which is explained below.

In the first execution interval, we start out with equal partitions for all the threads. At the end of the first interval, the previously described CPI based cache partitioning is used to partition the cache for the second interval. We do the same for the second interval, in the process collecting two data points for the CPI models. Later, at the end of each interval, we build a runtime CPI model for each thread. Specifically, we model the dependency of CPI on the number of cache ways. These CPI models are built at runtime, for each of the threads using the available data points. By data points, we mean the assigned number of cache ways and the corresponding CPI figures observed under these cache ways. Using these available data points for each thread, we use a simple cubic spline interpolation [80] to fit a curve for each of the threads. The choice of the curve fitting algorithm used is independent of the partitioning scheme, and therefore, any other algorithm

```
Initialization:

• Start out with equal partitions in the first interval.
      $\forall\ t,\ partition_i = \frac{TotalCacheWays}{NumberofCores}$

At the end of first two intervals:

• Use the previous CPI based cache partitioning.
    • Record CPI for each thread t, $CPI_i$.
    • Assign cache partitions to threads, based on their CPIs.
        Partition for thread t,
            $partition_t = \frac{CPI_t}{\sum CPI_i} \times TotalCacheWays$

At the end of each interval:

• Record the CPI value for each thread t, $CPI_i$.
• Build performance models for each of
the threads using cubic spline
    •Model CPI to cache way dependency
• Determine individual cache partitions by redistributing
cache ways based on the thread performance models
    •Step1: Reassign cache partitions
        $threadMaxCPI = HighestCPI$
        $threadMax$ = Thread with highest CPI
        $threadMinCPI = LowestCPI$
        $threadMin$ = Thread with lowest CPI
        $waysMaxCPI = waysMaxCPI + 1$
        $waysMinCPI = waysMinCPI - 1$
    •Step2: Recalculate the thread CPIs after
     reassignment (step 1) based on the individual
     thread performance models
        $newThreadMax$ = Thread with highest CPI
        IF($threadMaxCPI \neq tewThreadMaxCPI$)
            $waysMaxCPI = waysMaxCPI - 1$
            $waysMinCPI = waysMinCPI + 1$
            EXIT
        ELSE GOTO Step1
• Assign the newly calculated cache partitions to threads
```

Figure 5.10. Dynamic curve fitting based cache partitioning scheme.

could also be used. The CPI curves of the threads can be of any form. Now, the goal is to allocate ways so as to minimize the CPI of the highest CPI thread, which essentially minimizes the overall CPI of the application.

Since the search space for this problem is very large, we employ a heuristic strategy with minimal runtime overhead in our scheme. Our curve fitting algorithm tries to find the best possible cache way partition that minimizes the CPI of the highest CPI thread. Minimizing the CPI of the highest CPI thread is synonymous with speeding up the slowest thread (critical path thread), thereby speeding up the entire application execution.

As described earlier, at the end of each execution interval, the execution characteristics such as instruction counts and cycle counts are recorded for each thread

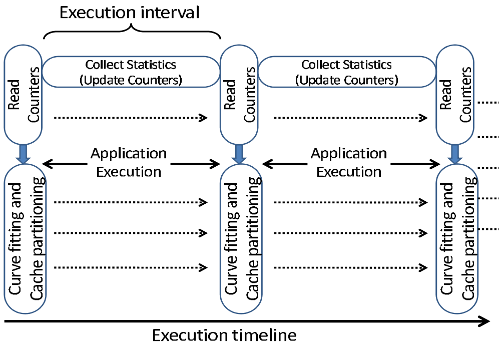

Execution interval
Collect Statistics (Update Counters)
Read Counters
Application Execution
Curve fitting and Cache partitioning
Execution timeline

Figure 5.11. Dynamic cache partitioning scheme in action during application execution.

along with the current cache partition. Using the current data point (cache ways, CPI) and the previously recored data points, a CPI model is built for each thread of the application being executed. Then, our cache partitioning algorithm is invoked. At each iteration of our partitioning algorithm, we take away a cache way from the thread with the lowest CPI (fastest thread) and assign it to the thread with the highest CPI (slowest thread). After this repartitioning, the CPI values are recalculated for all the threads based on the thread CPI models built earlier. Notice that, by recalculating the CPI values this way, whether the repartitioning has actually helped or not is taken into account. It is important to note here that, determining CPI values using the CPI models yields predicted CPI values for each of the threads and not the actual CPI values. After recalculating the thread level CPI values this way, repartitioning step is performed again, and so on. Thread CPI recalculation based on the thread CPI models and the cache way repartitioning steps are iteratively repeated this way. The termination point is when some other thread becomes the highest CPI thread. When that happens, we revert back the assignment by one step and terminate. Finally, we apply the calculated cache way assignment for the threads. Figure 5.10 shows a detailed sketch of the workings of our curve fitting based cache partitioning heuristic, and Figure 5.11 summarizes the entire procedure in a pictorial form.

5.4.3 Implementation Details

In our implementation, we consider an operating system (OS) cache allocator which allocates a certain amount of cache to each of the applications in a workload [64].

The application now executes under the control of our proposed runtime system which implements the cache partitioning algorithm. Implementing the cache partitioning this way (i.e., within a runtime system) gives us more flexibility. This is because it is not always easy to extract thread-level information in commercially available OSs. We envision a hierarchical system (shown in Figure 5.12), where OS manages the cache-partitioning among applications and the runtime-system manages the cache-partitioning among the threads of an application. Note that, in this setting, our intra-application scheme can be applied to each application simultaneously.

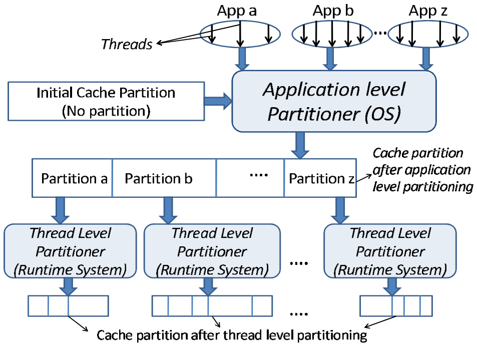


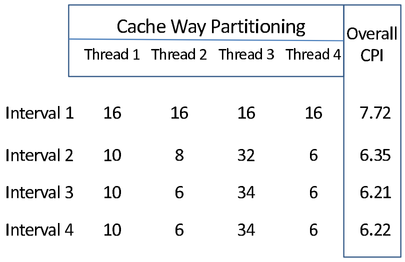
Figure 5.12. Hierarchical cache partitioning system.

5.5 Experimental Evaluation

We use the experimental setup described in Section 7.6.1 to evaluate our dynamic partitioning scheme. As mentioned before, we use Simics [57], which is a full system simulator to implement our scheme. We implement our cache partitioning scheme as a module in Simics. The cache partitioning scheme acts as a dynamic runtime system which implements the cache partitioning as described in Section 5.4.3. The runtime system decides the individual thread cache partitions and then issues the cache partition commands to Simics which actually partitions the shared cache. Runtime system collects the execution characteristics of the threads by reading the performance counters. The results presented below include all the runtime overheads incurred by our implementation.

5.5.1 Dynamic Cache Partitioning Snapshot

We now provide a brief snapshot of our dynamic cache partitioning scheme in action. Figure 5.13 shows the working of our dynamic cache partitioning scheme across a small set of execution intervals of NAS CG application [81]. During the first execution interval, the cache partition is equal. During the next intervals, cache is partitioned based on their run time cache models in order to speed up the critical path thread. In this example, since thread 3 is the slowest thread (the CPI values for threads 1, 2, 3, 4 were 3.06, 2.96, 6.35, 2.95, after the first interval), it is given the largest cache partition and consequently, as can be seen, the overall CPI of the application is reduced, thereby, improving the overall performance. A similar situation results as we move from interval 2 to interval 3. That is, our approach successfully modulates the cache space allocation dynamically during execution.



	Cache Way Partitioning				Overall CPI
	Thread 1	Thread 2	Thread 3	Thread 4	
Interval 1	16	16	16	16	7.72
Interval 2	10	8	32	6	6.35
Interval 3	10	6	34	6	6.21
Interval 4	10	6	34	6	6.22

Figure 5.13. A snapshot of our dynamic cache partitioning scheme in action across four consecutive execution intervals of the NAS CG application. This figure shows the cache ways allocated to each of the threads during the execution intervals and the resulting CPI values.

5.5.2 Comparison with Alternate Schemes

We start by comparing our dynamic cache partitioning scheme with a statically partitioned cache with equal partitions, which is the same as a private L2 cache. A private cache also yields the optimal fairness results. Therefore, comparison with private cache is the same as the comparison with fairness oriented schemes (such as those presented in [19] [17]). Figure 5.14 shows the performance improvement achieved by our dynamic cache partitioning scheme over a private, equally partitioned cache. Dynamically partitioned cache results in performance improvement of up to 23% over the private cache case. This is because our algorithm adapts dynamically to build performance models and allocates available cache space specifically to speed up the critical path thread, as opposed to a simple static partition. On average, our scheme outperforms the private cache configuration by about 11%.

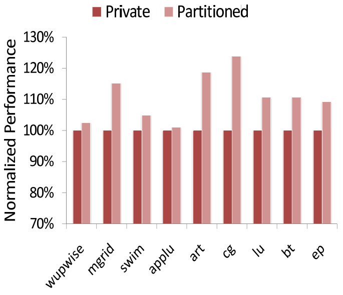


Figure 5.14. Performance improvement over an equally partitioned cache (private cache).

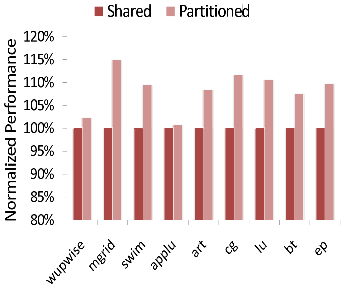


Figure 5.15. Performance improvement over a shared unpartitioned cache.

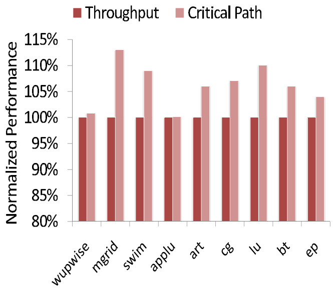


Figure 5.16. Performance improvement over a throughput-oriented cache partitioning scheme.

Next, we compare our proposed dynamic cache partitioning scheme to an unpartitioned shared cache (i.e., a fully-shared cache). Figure 5.15 shows the performance improvement achieved by our dynamic cache partitioning scheme over a shared, unpartitioned cache. The average performance improvement achieved is about 9%. Although shared cache is considered the most efficient in terms of utilization and data sharing, the dynamic cache partitioning scheme outperforms the shared cache. In three of the benchmarks we tested, dynamic partitioning scheme yields only a small benefit. This is because of the very small working set size of those benchmarks. We also compare our scheme to a throughput-oriented scheme. Here, we use the throughput oriented strategy employed by these prior schemes in the intra-application case for comparison with our scheme. As we can see from the plot in Figure 5.16, our proposed cache partitioning scheme outperforms the throughput-oriented scheme for all the applications we tested.

Chapter 6

Bandwidth Constrained Coordinated HW/SW Prefetching for Multicores

6.1 Introduction

Prefetching is a well-known memory latency hiding technique, which predicts future memory accesses and proactively fetches the corresponding memory elements to the cache ahead of time in order to hide memory access latencies during execution [82] [83] [84] [85]. Prefetching can either be implemented at the hardware level [82] [83] [85] [84] or by the software [86] [87]. The effectiveness of a prefetching scheme is directly dependent on the predictability of memory accesses, which is an application characteristic. In a multicore system, each core prefetches data elements independently into the cache. The benefits due to prefetching can potentially be different for different cores depending on the application characteristics. Further, each core/application can potentially be involved in both hardware and software prefetching. There have been previous techniques proposed to throttle inaccurate prefetchers and increase aggressiveness levels on more accurate ones [88]. Also, when the last level cache is shared, aggressive prefetching can worsen the cache interference problem, especially when it is inaccurate and/or inefficient. In such cases, it is helpful to throttle the prefetches that are inaccurate and cause high interference in the shared cache space [89].

In this chapter, we first study the comparative accuracies and benefits of software prefetching and different levels of hardware prefetching. We then study and analyze the impact of prefetching on the off-chip memory bandwidth performance. Prefetching can lead to increased off-chip bus traffic, and can potentially increase the pressure on the off-chip bandwidth. This can cause extensive bandwidth stalls.We explore the tradeoff between extensive aggressive prefetching and bandwidth stalls. Further, we study if the performance degradation due to bandwidth stalls wipe away the performance gains achieved as a result of prefetching.

We propose a hierarchical bandwidth-aware coordinated prefetching scheme that manages the prefetch aggressiveness levels of different cores such that the performance gains due to prefetching are improved, while the performance losses due to bandwidth stalls are reduced. This prefetch management scheme operates dynamically and decisions are made at the end of each execution interval. More specifically, a *global prefetch manager* considers the overall bandwidth delay and the prefetch effectiveness of each core during each execution interval, and then decides to increase or decrease the prefetch aggressiveness levels of the cores. This decision to change the prefetch levels of the cores is made such that the performance improvement due to prefetching is higher than the stall time due to limited bandwidth and contention. It then directs the individual core-level prefetch managers to change the prefetch levels correspondingly. At each core, a *core-level prefetch manager* manages and enforces the prefetch aggressiveness levels. This prefetch manager not only issues hardware prefetch requests but also handles the software prefetch instructions. It decides whether to allow software prefetching or hardware prefetching or both and also at what aggressiveness levels. It is to be noted that prefetching on a core can be termed very aggressive if both hardware prefetching at the highest aggressiveness level and software prefetching is enabled. Aggressiveness can be downgraded by reducing the aggressiveness of hardware or software prefetching or both. Overall, the main goal of our approach is to reward useful prefetchers and punish the ones that hurt bandwidth availability without any performance benefit. Lastly, we evaluate our proposed scheme on set of workloads comprising of applications from the SPEC 2006 benchmark suite [58] on a simulation based setup, and show that our scheme yields average system throughput benefits of about 8%, and up to about 10% over an off-chip bandwidth unaware scheme. To summarize, we make the following contributions in this chapter:

•We evaluate the performance benefit of both hardware (different levels) and software prefetching schemes. We later compare the performance improvement due to prefetching against the performance degradation due to the extra pressure it exerts on the off-chip bandwidth.

•We propose a *hierarchical prefetch management scheme* that tries to dynamically change the prefetch levels of the individual cores such that the performance degradation due to bandwidth contention is reduced and the performance improvement due to prefetching is improved.

•We present an extensive experimental evaluation of the proposed hierarchical prefetch management. Our results show that the proposed scheme is very effective in practice and improves the system throughput by up to 10%, and by an average of 8%.

6.2 Background and Methodology

6.2.1 Prefetching

Prefetching is a widely employed technique intended to improve on-chip cache performance [82] [83] [84] [85] [86] [87]. Prefetching, however, is not always beneficial. Some fraction of the predicted memory requests are never accessed. This is not the only instance of wasted prefetching. A future memory request prediction can turn out to be true but before the prefetched memory element is accessed, it might be evicted from the cache. Also, a prefetched request may kick out a useful data element from the cache. In these instances, prefetching increases the off-chip bus traffic and possibly cause bandwidth stalls without any significant benefit. Therefore, prefetch accuracy, which is an application characteristic determines the overall performance benefit from prefetching.

Hardware Prefetching. In the case of hardware prefetching, the future memory access prediction and the process of initiating requests to prefetch those elements are carried out by the hardware at runtime. Due to costs and limits on delay, hardware prefetchers generally implement a simple stride based prefetching or a stream based prefetching. A very aggressive hardware prefetcher would typically predict a large number of future memory requests and prefetch them. In comparison, a prefetcher with a lower aggressiveness level would be more conservative, predicting and issuing fewer prefetches. In this chapter, we refer to and implement a stream prefetcher [88] [90] [91]. Aggressiveness level of a stream prefetcher is defined by two parameters: prefetch distance and prefetch degree [88] [90] [91]. *Prefetch Distance* dictates how far ahead of the demand access stream the prefetcher can issue prefetch requests, and *Prefetch Degree* determines how many cache blocks to prefetch when there is a cache block access to a monitored memory region.

Software Prefetching. In this case, the future memory access prediction is made statically, at compile time or at the coding time, and specific instructions are inserted into the code body to prefetch those predicted elements at the time of execution. Some applications render themselves to easy compile time prediction in which case the software prefetching is very effective [86] [87] [92]. Software prefetching also has the ability to employ complex and time consuming prefetching algorithms since the process is done apriori at compile time. Hardware prefetching, on the other hand, employs simpler prediction mechanisms but does well where software prefetching fails to analyze the code, e.g., as in the case of pointer-based applications.

6.2.2 Experimental Setup

We model the off-chip memory bandwidth and implement the prefetching infrastructure for multicores using a Simics [57] based in-house module. The base system architecture simulated in our evaluations is a four-core multicore machine with a shared L2 cache and a shared off-chip memory bandwidth. The shared L2 cache is assumed to be a partitioned cache (i.e., its cache ways are distributed evenly across applications though in principle we could use any partitioning strategy). The cores simulated in this system are based on the UltraSparc 3 architecture [3]. The main architectural details of the simulated system are shown in the table given in Figure 6.1. In the evaluation of the proposed dynamic scheme later, we employ execution intervals of 10 million instructions. The hardware prefetcher used in this chapter is a stream prefetcher [88] [90] [91] with 64 streams per prefetcher.

Benchmarks. For all the motivational and evaluation purposes, we use the applications from the SPEC 2006 benchmark suite [58], and construct our workloads

Core architecture	UltraSparc 3, 3.1 GHz
Operating system	Sun Solaris 9
L1 caches	private, 3 cycle latency, direct-mapped
L2 cache	shared, 15 cycle latency, 16 way associative
Memory latency	260 cycles
Hardware Prefetcher	64 stream prefetcher per core, 4 prefetch levels
DRAM controller	demand-prefetch equal priorities, on-chip, 128 entry req buffer, FR-FCFS
DRAM chip	refer to Micron DDR2-800 [93]

Figure 6.1. Default system parameters used.

from the subsets of these applications. To enable software prefetching on the applications, they are compiled on a SUN compiler with the highest optimization flag set.

Terminology. In this chapter, by "prefetch level", we mean the "aggressiveness level" of the prefetcher. All types of prefetching mentioned in this chapter are implemented for the last level of cache in a multicore. Whenever we refer to "software prefetching" in this chapter, we mean the handling of the software-inserted prefetch instructions in the hardware. We do not propose or implement a new software prefetching algorithm. We compile the applications using a software prefetch enabled compiler that inserts prefetch instructions into the executable. We only refer to the way these instructions are handled in the hardware.

6.3 Empirical Motivation

6.3.1 Prefetching Benefits

The goal of this section is to compare the performance of various prefetching techniques with different aggressiveness levels across different applications.

Hardware Prefetching. Figure 6.2 plots the performance of our applications when different levels of prefetching are enabled compared to the case of no prefetching. We experimented with four different prefetch levels: *no prefetching, level 1, level 2* and *level 3*. *Level 1* prefetching has a prefetch distance of 4 and prefetch degree of 1.

Prefetch distance and prefetch degree of *level 2* are 16 and 2 respectively, and those of *level 3* are 64 and 4. In this set of experiments, software prefetching is disabled, which means the prefetch instructions are ignored as no-ops. Since we are

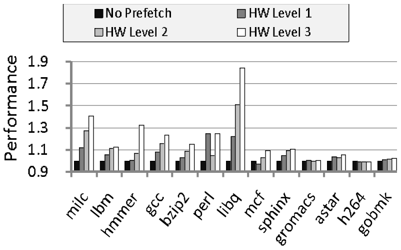


Figure 6.2. Performance comparisons of different levels of hardware prefetching. The performance values are normalized to that of the no prefetching case.

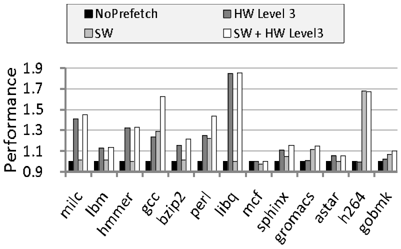


Figure 6.3. Performance comparisons of software prefetching, hardware level 3 prefetching, and both with the case of no prefetching. The performance values are normalized to that of the no prefetching case.

first interested in studying the performance benefits of prefetching in isolation, the performance effects due to bandwidth constraints are not considered in these experiments. From Figure 6.2, we can infer that while some applications are prefetch sensitive and, therefore benefit from more aggressive levels of prefetching, others do not exhibit large performance gains as prefetch levels are increased. In the above scenario, the prefetch levels can be reduced on applications that are not very prefetch-sensitive without a high performance penalty. On the flip side, increasing the prefetch levels on prefetch-sensitive applications can be very beneficial.

Software Prefetching. Figure 6.3 compares software prefetching, hardware level 3 prefetching, a combined software hardware prefetching scheme against the no prefetching case. One can see from this plot that, for some applications, hardware prefetching does much better than software prefetching, whereas for some others, it is the other way around. More interestingly, in some cases, enabling both hardware and software prefetching is much better than enabling just one of them, as in the case of *gcc* and *perl*. In some other cases, although effective individually, enabling both does not do any better than enabling only one of them, as in the case of *astar* and *h264*. Therefore, in a multicore system, some applications perform better when both hardware and software prefetching are enabled, while some others perform equally well with just one of them enabled.

6.3.2 Off-Chip Bandwidth Effects

In this section, we study the effect of prefetching on off-chip bandwidth pressure. We employ an off-chip bandwidth of 6.4 GB/s in these experiments. For this purpose, we selected a workload of four applications: *lbm, mcf, libquantum,* and *milc.* These four applications are executed on a four-core processor (one application per core) with a shared, partitioned cache, and a shared off-chip bandwidth.

One core prefetching. In the first run, we enabled prefetching only on the first core which executed *lbm*, while disabling prefetching on all other cores. We experimented with software prefetching, three levels of hardware prefetching, and a combined hardware-software prefetching scheme. The results in Figure 6.4 show that *lbm*, which executed on core 1, achieves a performance benefit when compared to the case of no prefetching. However, its benefits are reduced due to the limited bandwidth constraint. Further, it degrades the performance of the other applications due to the additional requests (prefetch requests from core 1) and the resulting bandwidth stalls. When using the most aggressive prefetching, the performance degradations on the other cores are significant.

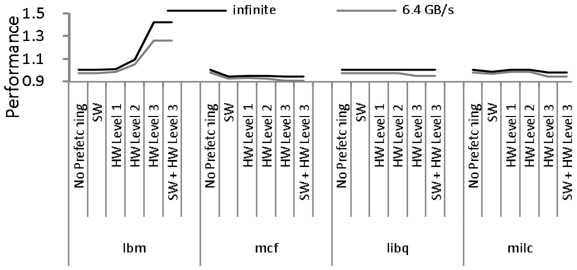


Figure 6.4. Performance comparisons of different prefetching schemes with both the infinite bandwidth case and a bandwidth of 6.4 GB/s, when prefetching is enabled only on core 1 (*lbm*) and disabled for all others.

We also repeated this by enabling prefetching on core 2, core 3 and core 4 alone, and observed similar results. Therefore, prefetching can have different degrees of performance degradation due to bandwidth constraints. Further, *aggressive prefetching by one core can adversely impact the performance of other applications due to bandwidth contention and the resulting delays.*

All cores prefetching. We also considered a more realistic execution scenario, where different applications prefetch memory elements individually and the cores share the available off-chip bandwidth. In this case, prefetching is enabled on all the cores. In Figure 6.5, we plot the comparative contributions to the bus traffic by the applications when prefetching is enabled on all the cores. The bus traffic increases rapidly when the prefetch level is increased for some applications, while for others, the increase is not that steep (for instance *milc*).

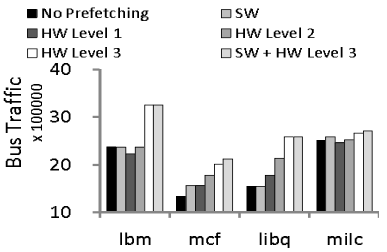


Figure 6.5. Contributions to the bus traffic by different applications.

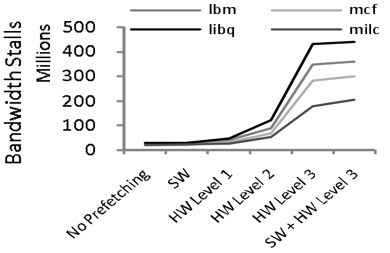


Figure 6.6. Bandwidth stalls (in cycles) suffered by applications as the prefetching level is increased.

Figure 6.6 shows how this increase in bus traffic translates into stalls due to limited bandwidth. Note that, even if the bus traffic increase is small, bandwidth stalls can be significant. The above two graphs plot absolute values of bus traffic increase and bandwidth stall cycles. Figure 6.7, on the other hand, illustrates how these factors affect the performance of applications when different prefetch variants are enabled for both the infinite bandwidth case and a more realistic case of 6.4 GB/s bandwidth.

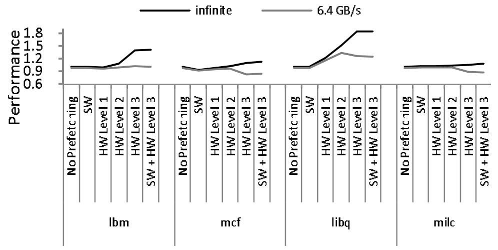


Figure 6.7. Performance comparisons of different prefetching schemes with both the infinite bandwidth case and a bandwidth of 6.4 GB/s, when prefetching is enabled on all cores.

In the limited bandwidth case, prefetching aggressively in a bandwidth unaware manner on all the cores results in some performance improvement only on core 3 (*libq*). In all other applications/cores, performance degradation due to limited bandwidth completely wipes out all the benefits from prefetching and in some cases results in a net performance degradation. This effect increases with increasing prefetch levels. Also, for some applications, while absolute values of bandwidth stalls in Figure 6.6 increase sharply with prefetch levels, performance degradation is not that steep. Therefore, some applications are more bandwidth-stall resistant (tolerant). In modeling the performance effects later in Section 6.4.3, we take this into account. We do not just consider prefetch accuracies and the resulting bus traffic as the basis as done previously [88] [89] but also consider the bandwidth stalls and the actual impact of bandwidth stalls on application performance as the basis.

To summarize, while prefetching aggressively can improve performance, it can also hurt the performance due to bandwidth constraints. Therefore, it is important to enable prefetching without increasing bandwidth delays extensively.

6.3.3 Prefetch Request Priority

Increase in bandwidth delays due to prefetching typically occurs only if prefetch requests are treated on par with demand memory requests. If normal load/store (demand) memory requests have a higher priority than the prefetching requests, then additional bus traffic due to prefetch requests may not lead to any additional bandwidth delay. It is to be noted here that bandwidth delays might still be present in the system but those delays are due to the normal (demand) memory requests, and will be present irrespective of whether prefetching is turned on or not.

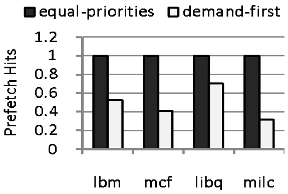


Figure 6.8. Comparison of equal priorities for prefetch and demand requests versus a scheme where demand requests are prioritized over prefetch requests in terms of the number of useful prefetches.

Prioritizing demand requests and prefetching requests equally leads to increased performance improvement from prefetching as can be seen in Figure 6.8. This is due to the fact that if the prefetch requests have a lower priority than the demand requests, then the prefetch requests can get delayed inordinately and these increased bandwidth delays can render most of prefetch requests useless (since prefetched data would be brought into the cache late). This leads to decreased prefetch efficiency and, therefore, decreased positive performance impact of prefetching [94]. Therefore, our proposed scheme employs equal priorities, and tries to keep the number of useful prefetches high, while at the same time, mitigating the additional bandwidth stalls due to prefetch requests.

6.4 Bandwidth Aware Prefetching

Figure 6.9 summarizes the operation of our proposed scheme. A *global prefetch manager* makes decisions on whether to increase or decrease the prefetching levels on the individual cores and the decisions are communicated to the *core-level prefetch manager*.

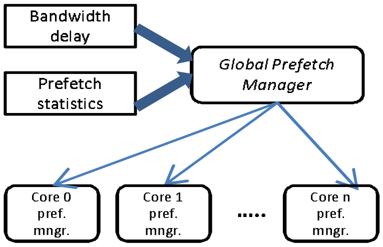


Figure 6.9. Hierarchical bandwidth aware prefetching scheme that includes a *global prefetch manager* and a set of *core-level prefetch managers*.

The details on how these decisions are made are presented in Section 6.4.3. After the global manager directs a core-level prefetch manager to either increase or decrease the prefetch level of the core, the core-level manager applies the prefetch-level changes locally (i.e., to the core it is attached to), as described in Section 6.4.1. This prefetch management scheme works dynamically, making decisions on prefetch level changes and applying those changes at the end of each execution interval. This scheme is also history based, in the sense that all the relevant statistics, which include the total bandwidth stall-time and the prefetch efficiency counters of individual cores, collected during an execution interval are used to make decisions for the next execution interval.

Implementation. Hardware support is needed to maintain the performance counters. The prefetch management scheme itself is implemented in the runtime system/OS, which reads these hardware performance counters.

6.4.1 Core-Level Prefetch Manager

The core-level prefetch manager sets and enforces the prefetch aggressiveness level at the core level. It can either increase or decrease the prefetch level based on the directions from the global prefetch manager.

The core-level prefetch manager handles the changes in prefetch levels of the hardware prefetcher similar to that proposed in [89]. In addition to the hardware prefetcher, our proposed prefetch manager also employs a *software prefetcher*, which is an engine that handles all the software prefetch instructions issued by the core (compiler-inserted or programmer inserted). A prefetch instruction, when issued, results in a prefetch request. All such prefetch requests are routed through our proposed software prefetcher. When the global prefetch manager directs the core-level manager to either increase or decrease the prefetch level, the core-level manager can increase or decrease the prefetch level of either the hardware prefetcher or the software prefetcher. What we mean by "prefetch levels" in hardware and software prefetchers is explained later in detail. The role of the core-level prefetch manager in controlling the prefetch levels of both hardware and software prefetches is illustrated in Figure 6.10. The global prefetch manager either increases or decreases prefetch level, and does not set absolute values.

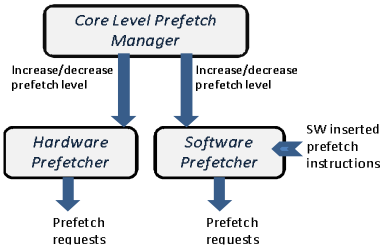

Core Level Prefetch Manager
Increase/decrease prefetch level
Increase/decrease prefetch level
Hardware Prefetcher
Software Prefetcher
SW inserted prefetch instructions
Prefetch requests
Prefetch requests

Figure 6.10. Details of a core-level prefetch manager, which controls the prefetch levels of both hardware and software prefetchers of a core.

```
increase_prefetch_level()
 begin
   $accuracy_{HW} = \frac{prefhits_{SW}}{prefetches_{HW}}$
   $accuracy_{SW} = \frac{Prefhits_{SW}}{prefetches_{SW}}$
   if $accuracy_{HW} > accuracy_{SW}$
     //Increase HW prefetch level
     increase $prefetch\_distance_{HW}$
     increase $prefetch\_degree_{HW}$
   else
     //Increase SW prefetch level
     increase $prefetch\_distance_{SW}$
     increase $prefetch\_degree_{SW}$
 end
```

Figure 6.11. Prefetch level increase function.

The decision of whether to change the prefetch level of the hardware prefetcher or the software prefetcher is determined by calculating the corresponding *prefetch accuracies*. More accurate prefetcher is always preferred. This way, we prioritize either hardware or software prefetching based on their accuracies (the prefetch increase function is shown in Figure 6.11, prefetch decrease function is on similar lines).

6.4.2 Prefetch Levels

Hardware Prefetch Levels. We implement a stream prefetcher for hardware prefetching [90]. As mentioned earlier, the aggressiveness level of a stream prefetcher is defined by two parameters: *prefetch distance* and *prefetch degree*. Our hardware prefetcher design is similar to that implemented in [88] and further details on implementation can be found in [88] [90] [91]. In essence, the prefetch distance determines how far ahead of the memory stream the prefetch requests are issued and the prefetch degree determines how many prefetch requests are issued each time. We implement four prefetch levels in this work: *no prefetch*, *low prefetch*, *medium prefetch*, and *high prefetch*. No prefetch level performs no prefetching. Low prefetch level performs prefetching with a prefetch distance of 4 and prefetch degree of 1. Medium prefetch performs prefetching with a prefetch distance of 16 and a prefetch degree of 2, while the high prefetch level has prefetch distance of 64 and prefetch degree of 4.

Software Prefetch Levels. The software prefetcher implements the software prefetch levels by filtering the prefetch requests. As mentioned before, the software prefetcher receives all the prefetch requests that are issued by the software (compiler inserted or programmer inserted) instructions. The four aggressiveness levels of software prefetching are: *no prefetch*, *low prefetch*, *medium prefetch*, and *high prefetch*. When the level is set to no prefetch, all the prefetch requests are dropped. In the case of low prefetch level, two in every four prefetch requests are dropped, while only one in every four is dropped in the case of medium prefetch level. When the level is set to high prefetch, all prefetch requests coming from the software inserted prefetch instructions are issued by the software prefetcher without dropping any of them.

6.4.3 Global Prefetch Manager

As shown in Figure 6.9, the two main inputs to the *global prefetch manager* are the total bandwidth stall-time and the prefetch statistics.

Bandwidth stall-time. A demand request stalls in the memory controller queue if there are other requests ahead which are being serviced or waiting to be serviced. While the prefetch requests may also wait, they do not contribute to performance degradation (a higher wait-time for prefetch requests can of course limit the benefits due to prefetching). We define "*bandwidth_stall*" as the total stall-time (in cycles) experienced by the demand requests in a given execution interval. It is the sum of all individual demand request stall-times in that execution interval. Observe that "stall-time" in this context refers to wait-time in the queue due to bandwidth constraint. It does not include the time for a demand request to get serviced (to perform the memory operation). We compute *bandwidth_stall* using a simple counter in the memory controller. Since the off-chip bandwidth is a single resource shared across all the cores, *bandwidth_stall* is a single value, which is the sum of bandwidth stalls of all requests of all cores serviced by the off-chip bandwidth during the given execution interval.

Prefetch Statistics. As described in Section 6.4.1, each core has a hardware prefetcher and a software prefetcher associated with it. We define "$prefetches_i$" to be the total number of prefetches issued by core i. It is the sum of the number of prefetches issued by the hardware prefetcher and those issued by the software prefetcher. The metric "$prefhits_i$" is defined as the total number of prefetch requests (both hardware and software) that turned out to be hits for core i. These values are calculated using the prefetch bit of the cache line and by employing counters in the prefetchers.

Benefit Estimation. The performance improvement on core i due to prefetching is quantified by a parameter called "$benefit_i$". This improvement is specifically due to the avoidance of a fraction of core i cache misses. The metric $benefit_i$ is computed for each core i using the prefetch statistics collected during the execution interval as follows: $benefit_i = \frac{Reduction\_in\_cache\_miss\_stall\_time_i}{instructions_i}$.

Therefore, accounting for this reduction in cache misses, we obtain:
$benefit_i = \frac{(misses\_old_i - misses\_new_i) \times avg\_miss\_penalty}{instructions_i} = \frac{prefhits_i \times avg\_miss\_penalty}{instructions_i}$,
where $instructions_i$ is the number of instructions executed in the current execution

interval, $misses\_old_i$ is the estimated number of cache misses if prefetching was not enabled, $misses\_new_i$ is the number of cache misses with prefetching, and $avg\_miss\_penalty$ is the average cache miss penalty in cycles.

Cost Estimation. Prefetching leads to additional memory requests (in addition to the normal load/store demand requests). The measure of performance degradation suffered by core i due to memory bandwidth stall-time resulting from these prefetch requests it issues is quantified by the metric $cost_i$. Due to the fact that memory bandwidth is shared, the additional prefetches issued by core i can cause bandwidth stalls for not only core i but also for all other cores as well. As a result, $cost_i$ should take all these stalls into account. Firstly, the total bandwidth stall caused by the prefetches issued by all the cores can be estimated as below: $total\_prefetch\_stall = \frac{\Sigma_{i=0}^{n} prefetches_i}{total\_requests} \times bandwidth\_stall$, where $\Sigma_{i=0}^{n} prefetches_i$ is the sum of prefetches issued by all the cores during the interval, $total\_requests$ is the total number of requests that reached the memory controller during the execution interval (i.e., sum of the demand and prefetch requests), and $bandwidth\_stall$ is the total bandwidth stall time as defined earlier. We can now estimate the stall caused by core i (due to the prefetches issued by core i) as follows: $prefetch\_stall_i = \frac{prefetches_i}{\Sigma_{i=0}^{n} prefetches_i} \times total\_prefetch\_stall$. For each core i, we now have $prefetch\_stall_i$, which is the estimated absolute bandwidth stall-time caused by the prefetch requests issued by core i. Since the off-chip bandwidth is a shared resource, $prefetch\_stall_i$, caused by core i can affect demand requests of any of the cores. We define $band\_stall_{i,j}$ as the bandwidth stall caused by the prefetches from core i on the performance of core j (on the demand requests of core j). This value estimates the fraction of the bandwidth stall of core j, due to the prefetch requests issued by core i. We can estimate $band\_stall_{i,j}$ as follows: $band\_stall_{i,j} = \frac{demand_j}{\Sigma_{i=0}^{n} demand_k} \times prefetch\_stall_i$, where $demand_j$ is the total number of demand requests issued by core j, which in this case is approximately equal to the number of L2 cache misses on core j, $\Sigma_{i=0}^{n} demand_k$ is the total number of demand requests issued by all cores.These $band\_stall_{i,j}$ values estimated above are the absolute stall times in cycles and not the impact on performance. Therefore, we now estimate $cost_i$, which is a measure of the total performance degradation caused by the prefetches issued by core i on the performance of all cores including core i. Note that performance degradation considered above is just the effect of bandwidth stalls. The value of $cost_i$ can be estimated as follows: $cost_i = \Sigma_{j=0}^{n} \frac{band\_stall_{i,j}}{instructions_j}$. It is important to note that, we do not consider prefetch accuracies or the absolute bandwidth stalls in our estimation of $benefit_i$ and $cost_i$ values. We estimate both these values in terms of the net effect on the application performance.

Algorithm. The global prefetch manager manages the prefetch levels for each core with the goal of improving the overall performance gains due to prefetching. In order to do so, global manager employs a cost/benefit analysis based scheme

```
$global\_prefetch\_manager()$
$begin$
 for each $execution\ interval$:
   read $bandwidth\_stall$
   for each $i$ from 0 to $num_{c}ores$:
     read $instructions_i$, $prefetches_i$ and $prefhits_i$
     compute $benefit_i$ and $cost_i$
     if $(benefit_i - cost_i) >= cost_i \times \alpha$ then
       //increase the prefetch level of core $i$
       $core\_level\_manager\_i.increase\_prefetch\_level()$
     else if $(benefit_i - cost_i > 0$ $and$
     $benefit_i - cost_i < cost_i \times \alpha)$
       //do not change the prefetch level of core $i$
     else $(benefit_i - cost_i) <= 0$ then
       //decrease the prefetch level of core $i$
       $core\_level\_manager\_i.decrease\_prefetch\_level()$
 end for
$end$
```

Figure 6.12. The algorithm executed by the global prefetch manager.

A prediction based dynamic scheme is employed by the global manager, i.e., the algorithm works by computing and making prefetch level changes for cores at the end of each execution interval. This algorithm is shown in Figure 6.12. To begin with, all cores prefetch at the highest aggressiveness levels. The $benefit_i$ and $cost_i$ values are estimated for every core i at the end of each interval after reading the relevant performance counter values. For each core i, the prefetch level is increased if the $benefit_i - cost_i$ is greater than the $cost_i \times \alpha$ (i.e., if $benefit_i$ is greater than $cost_i$ by α percentage). If, on the other hand, the $benefit_i - cost_i$ is lower than the $cost_i \times \alpha$ but greater than zero, then the prefetch level is left unchanged. Finally, if $benefit_i$ is less than the $cost_i$ value, then the prefetch level is decreased for core i. The global prefetch manager enforces the prefetch level change for a given core i by directing the core-level manager of the corresponding core. The reason for reducing the prefetch level for a given core is obvious since the estimated benefit is lower than the estimated cost. On the other hand, increasing the prefetch level is more nuanced. The level is increased only if the estimated benefit is greater than the cost by a *pre-defined threshold value* (α).If the benefit is not greater than the cost by α percentage, the prefetch level is left unchanged. This algorithm can reduce the prefetch level of a core i gradually to zero (which means no prefetches are issued) when $benefit_i$ continues to be lesser than $cost_i$ after continuous prefetch level decrements. In this case, when the prefetch level is zero, $benefit_i$ will always be zero and the prefetch level will potentially be stuck at zero without being increased. To avoid this scenario, the core-level prefetch manager increments the prefetch level of a core to level 1 if the prefetch level is stuck at zero for more than two execution intervals. In this algorithm, since we consider benefit and cost values in terms of estimated changes in application performance, the goal is always to improve the performance of applications and improve the overall system throughput.

α values. The α values are tunable to make the prefetching scheme more conservative or more aggressive. We experimented with a lot of α values and finally determined that a value of 0.2 is reasonable. Therefore, in our implementation, if the benefit exceeds the cost by 20%, we increase the prefetch level.

6.5 Experimental Evaluation

Our evaluation setup is described in Section 7.6.1. A four-core machine with a shared, partitioned L2 cache was modeled as the underlying multicore architecture. We built several workloads that consist of four applications, each from the SPEC 2006 suite [58]. In all our evaluations, we collect results and data for a period of 1 billion cycles. Cache is however warmed up for a period of 500 million instructions prior to collecting results. We consider execution intervals of 10 million instructions. Our proposed prefetching scheme is called *Dyn_Band* throughout the experimental section.

Average Throughput. Figure 6.13 presents the throughput gain acheived by our proposed scheme (*Dyn_Band*) over other prior prefetching schemes when averaged over 10 different workloads we experimented with. Different workloads might benefit differently from the prior prefetching schemes. Our proposed scheme recognizes this and enables only those prefetching schemes and levels that benefits the workloads, also taking into account the bandwidth pressure exerted by the extra prefetch memory requests. Our proposed scheme yeilds an average system throughput gain of about 8% over the best of the previous prefetching schemes.

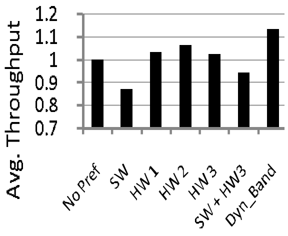


Figure 6.13. Throughput comparison averaged across multiple workloads.

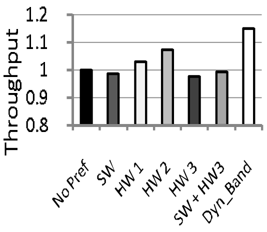


Figure 6.14. Throughput comparison for the workload (*lbm*, *mcf*, *libquantum*, and *milc*).

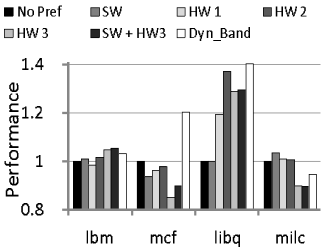


Figure 6.15. Performance comparisons of the applications in the workload (*lbm*, *mcf*, *libquantum*, and *milc*).

Workload Instance. In order to understand our proposed scheme in more detail, we now present the results for a single workload instance that consists of *lbm*, *mcf*, *libquantum*, and *milc*. The corresponding throughput results are shown in Figure 6.14. In this case, our proposed dynamic bandwidth-aware prefetching scheme improves throughput by 15% over the no prefetching scheme. Among the other prefetching schemes, hardware level 2 prefetching does better than others because of lower pressure on off-chip bandwidth. Our dynamic bandwidth-aware scheme has a throughput gain of about 8% over this hardware level 2 prefetching. Figure 6.15 shows the individual application performance values. We observe that the application *milc* gains about 40% in performance over the no prefetching scheme and *mcf* gains about 20%.

Dynamics of the system. In order to analyze the working of our proposed scheme, we consider the execution of a workload comprising of *bzip2, libq, sphinx* and *gromacs* applications, and focus on the performances of *libq* and *gromacs*. We track how our scheme works dynamically, and adjusts the prefetch levels of these two applications based on their $benefit$ and $cost$ values (note here that our scheme works and adjusts the prefetch levels of all four applications; we focus on just two for clarity). Figures 6.16 and 6.17 plot the observed $benefit$ and $cost$ values for these two applications for 11 execution intervals, when our scheme is used. In the case of *libq*, the $benefit$ value is consistently higher than the $cost$ value, while in the case of *gromacs*, the values are very close together.

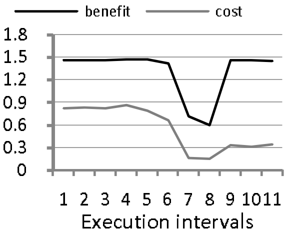


Figure 6.16. Benefit and cost values of *libq* during execution.

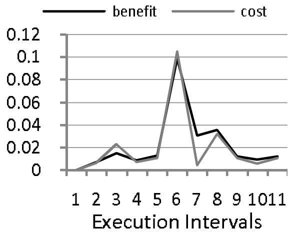


Figure 6.17. Benefit and cost values of *gromacs* during execution.

In order to study how our scheme dynamically changes the prefetch levels in accordance with the above values, we plot the $\frac{benefit-cost}{cost}$ values for the two applications for the same 11 execution intervals in Figure 6.18. Recall that, in the global prefetch management algorithm presented earlier in Figure 6.12, the equation $benefit_i - cost_i > cost_i \times \alpha$ is used to decide whether to increase the prefetch level or not. If the value $\frac{benefit-cost}{cost}$ is greater than α (0.2), then the prefetch level is increased and so on.

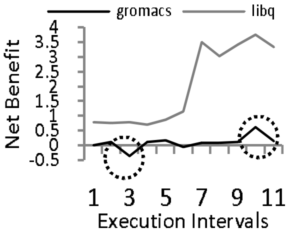


Figure 6.18. Net benefit values of *libquantum* and *gromacs* during execution.

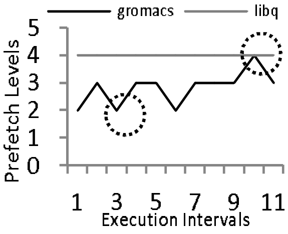


Figure 6.19. Prefetch levels of *libquantum* and *gromacs* during execution.

Figure 6.19 plots the prefetch level changes made by our proposed scheme for both the applications. Note that, at execution interval 3, the prefetch level of *gromacs* is reduced to 2 because the $benefit - cost$ value is less than zero (circled in Figures 6.18 and 6.19). Also, at execution interval 10, when $\frac{benefit-cost}{cost}$ becomes greater than 0.2 for gromacs, the prefetch level is increased to 4. However, it is reverted back because it was not highly benefitial. On the flipside, the prefetch level of *libq* is maintained at 4 since its $\frac{benefit-cost}{cost}$ values are consistently greater than 0.2.

Sensitivity analysis. We increased the memory bandwidth from 6.4 GB/s to 12.8 GB/s and executed the workloads. An average throughput improvement of about 7% over the best other prefetching scheme was observed. Therefore, even with higher bandwidth, our scheme achieves significant throughput improvement. We also experimented with different α values and found that a value of 0.2 provides the right balance.

6.6 Related Work

Hardware Prefetching. Hardware-controlled prefetching is an efficient way to implement prefetching [95] [82] [83] that tries to mitigate the negative effect of cold misses. Sequential prefetching automatically prefetches several consecutive data blocks into the cache upon a miss in the cache [84] [85]. Palacharla and Kessler investigate advanced stream buffers and filtering techniques to enhance the prefetching efficiency [91]. Hur and Lin discuss a dynamic stream detection technique that adapts the aggressiveness levels of prefetching in order to improve prefetching performance [96].

Software Prefetching. Seminal work related to software prefetching was authored by Mowry et al in [86], where they propose to use software controlled prefetch instruction insertion to enable prefetching. Other software prefetching schemes include [87] [86].

Prefetch Control. Srinath et al propose to use feedback control to improve the positive impact of prefetching and mitigate the adverse impact of harmful prefetches [88]. In [89], Ebrahimi et al investigate a control mechanism that can dynamically adjust the prefetch aggressiveness levels.

Off-Chip Bandwidth Studies. Rixner et al [22] introduce a scheduling policy that favors requests that hit in the row buffer over other requests. Nesbit et al suggest to prioritize memory requests of applications in accordance to their QoS requirements [29]. Rafique et al propose to adaptively change the fraction of memory bandwidth allocation for each thread [35]. In [97], Ipek et al study a machine learning approach in which a reinforcement learning based scheme is used to dynamically adapt scheduling decisions in the memory controller. Mutlu and Moscibroda proposed a stall time fair memory access scheduling in [21] and a parallelism-aware batch scheduling scheme in [28]. Liu et al study the effects of memory bandwidth partitioning on system performance [98].

Prefetching and Off-Chip Bandwidth. Lee et al propose to dynamically increase and decrease the priorities of prefetch requests at the memory controller in order to improve the benefits due to prefetching and decrease the penalties of inaccurate prefetchers [94]. In [99], Ebrahimi et al introduce a cooperative hardware/sofwtare approach to prefetch linked date structures in a bandwidth-efficient way.

In this chapter, we considered the off-chip bandwidth as an important constraint, based on which, the prefetching levels of different cores are adjusted such that the prefetch benefits are improved. We considered the off-chip bandwidth stalls instead of the inter-core interferences [89] as the constraint. We did so because inter-core interferences are not prefetch specific and can result from demand accesses as well. We also modeled the benefits and costs of prefetching in terms of performance changes in this work, which makes our scheme throughput driven, and evaluated the comparative benefits of hardware and software prefetching.

6.7 Concluding Remarks

In this chapter, we proposed a smart prefetch management scheme that exploits the performance benefits of prefetching while mitigating the performance degradation due to bandwidth stalls. Our proposed scheme is very effective in practice yielding a performance benefit of up to 8% in throughput over a bandwidth unaware prefetching strategy.

Chapter 7

Communication Based Proactive Link Power Management

7.1 Introduction

In NoC based multicore systems, a major contributor to chip power consumption is the NoC infrastructure. We found that, the NoC framework is responsible for as much as nearly 30% of the total chip power consumption. Communication links form a significant part of an NoC framework and their count increases with the number of cores in a CMP. This calls for power-aware design and power saving schemes which target not only power efficient cores but also power efficient link usage. Since, with the increase in the number of cores and with a similar increase in the number of communication links, possibility of more links being inactive increases dramatically, there is a need for a scalable power saving scheme which can exploit this effectively. Although circuit level and localized techniques are effective to an extent, they are not proactive, and therefore, lose out on important power saving opportunities. In this chapter, we propose a completely proactive scheme aimed at link power management.

We propose that execution of a multi-threaded application on an NoC based CMP can be characterized into phases based on the similarity across inter-core communication patterns. In this context, by communication pattern, we mean the usage of communication links in the system during execution. In case of a shared NUCA cache [100], which we consider, this usage of communication links is due to shared cache accesses and corresponding coherence traffic. The present circuit-level and localized schemes do not use this high level phase characterization information in their link power management. We propose to use the aforementioned phase characterization to implement a Markov based prediction scheme, which predicts the link usage of the next interval. This prediction can be used by a proactive link power management scheme to turn off predicted unused links and also to turn on links that are predicted to be used. The key advantage of this scheme is that, the links that are predicted to be used can be turned on ahead of time such that the turn-on latency is hidden and the performance remains unaltered. We show that this prediction based power management scheme can be very beneficial in reducing link energy consumption. We also note that this power saving scheme is remarkably scalable and can achieve increased power savings with increase in the number of on-chip cores and communication links. One of the important goals of our scheme, apart from minimizing energy consumption, is also to minimize the adverse impact on performance. We later show that, our scheme is very accurate in predicting link usage and hence has almost negligible performance impact.

Finally, we present the reduction in energy consumption, which is about 40% for two of the applications. We also present the average energy savings we achieve, which is about 23.5%.

7.2 Target Architecture

We consider a two-dimensional mesh based NoC that connects the nodes of a CMP, although our approach is equally applicable to other NoC structures. In this architecture, each node (core) has a private level 1 (L1) cache. On the other hand, the level 2 (L2) cache is shared among all the cores and is banked with each core containing an L2 bank. Figure 7.1 shows a 4×4 mesh structure we use to convey our idea. Most of the time, unless otherwise mentioned, we consider this 16 core, 4×4 mesh based CMP with a shared L2 cache which is 16 banked with each of the 16 cores containing an L2 bank. We use a static NUCA [100] scheme in this work although our scheme can be similarly used with dynamic NUCA [100] as well. We would like to emphasize that, in this chapter, by “inter-core communication”, we always mean an access made by a core to some other core’s L2 bank.

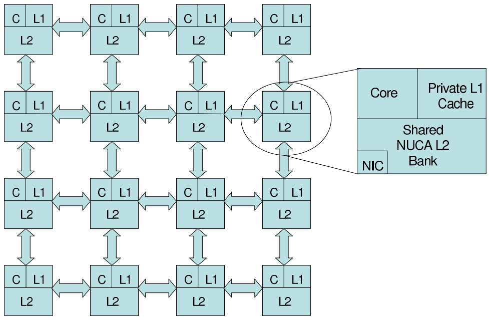
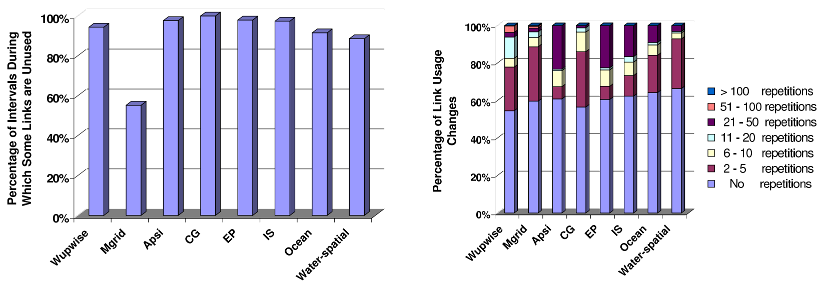


Figure 7.1. A 4×4 mesh NoC based CMP. Note that this is a block diagram and not the actual layout, and the routers are not shown for clarity.

7.3 Empirical Motivation

For any scheme aimed at link power savings to succeed, there should be considerable periods of execution during which some links are unused. If a multi-threaded application executing on an NoC based CMP uses all of the communication links during the entire period of execution, then any scheme aimed at saving link power will have limited returns. Fortunately, that is not the case in real applications. We profiled several parallel benchmarks from the SPEC OMP [58], NAS [81] and Splash2 [101] benchmark suites running on a 4×4 mesh architecture described in Figure 7.1. Profiling is done such that the execution is broken down into intervals of 5000 instructions, and links used during these intervals are recorded at the end of each such interval. We computed the percentage of such intervals during which at least some of the links in the interconnect network are not in use. Figure 7.2(a) shows our profiling results. As can be observed clearly, during a large percentage of intervals, at least some links are unused. Specifically, on average, in only 10% of intervals, all communication links are used. This is due mainly to the data allocation and the resulting cache bank access pattern exhibited by a program execution as we show in the next section. We also observed that the percentage increases slightly if the instruction interval is shortened. The number of links that are unused in such intervals determine the “window of opportunity”, which in other words, means the amount of power savings that can potentially be extracted. The profiling results above serve as the key motivating factor for the scheme we propose in the coming sections.



(a) (b)

Figure 7.2. Left - Percentage of intervals during which at least a few links are unused. We see that, on average, in only about 10% of intervals, all links are used. Right - shows the number of intervals, a new link usage pattern lasts (repeats) before it changes again to a different usage pattern.

Another key factor which needs to be considered is the "repetitive phase behavior" and hence possible "predictability" in parallel application's link usage. During execution, every time a new link usage pattern occurs, an important question is how long does that link usage pattern last before it changes again. Figure 7.2(b) shows the distribution of the number of times a link usage pattern repeats before there is a change. On average, after 10% of link usage changes, link usage remains the same for 21 to 50 intervals. After 3% of link usage changes, the usage pattern remains the same for 11 to 20 intervals; after 6.6% changes, the same usage remains for 6 to 10 intervals and after 19.1% changes, 2 to 5 times. Overall, on average, whenever a new link usage pattern arises, on nearly 40% of occasions, it remains for more than one interval before it changes again. It is important to note that, we are talking about instruction intervals (intervals of 5000 instructions) here and hence the link usage pattern repeating twice implies that the link usage remains the same for 2×5000, which is for 10,000 instructions. This is an important statistic which hints at repetitiveness and predictability in link usage patterns and possible success of predictive schemes. As we show in the next section, this predictability and repetitiveness results from the data allocation and the cache bank access pattern exhibited by the program execution.

7.4 Link Usage Based Phase Classification

Repetitive behavior is an execution characteristic of most applications. This repetitive behavior can be on the basis of similarity in the basic blocks touched or on the basis of similarity in performance metrics such as cache misses [102]. We use inter-core communication as the basis for characterizing the program execution into phases. Therefore, we classify the execution intervals into phases based on communication link usage. Each execution interval is an interval of 5000 instructions in our classification scheme. Since communication pattern is an application characteristic, instruction interval can be customized for an individual application by using profiling results. Although this interval length can be configured and further tuned as mentioned above, we found that, an interval of 5000 instructions works well for all applications we tested since it captures the repetitive behavior in inter-core communication pattern well. The usage pattern of communication links during execution depends on the data allocations and the data access patterns exhibited by the application, which manifests itself as L2 bank accesses. This means that, as the execution of a parallel application progresses, the L2 cache accesses and hence the communication link usage goes through phases. In this work, we represent the communication link usage in the form of a vector called "Link Vector", and carry out our phase characterization using this novel concept.

7.4.1 Link Vector

We represent the state of all the links in our NoC in the form of a link vector. Each bit in a link vector represents a link in the NoC and there is bit for every link. Consequently, the number of bits in the link vector is the same as the number in links in the on-chip network. Bit value 1 implies a used state, which means the link is being exercised, and a bit value of 0 implies an unused state, which means the link is idle. For example, in the case of NoC illustrated in Figure 7.1, the corresponding link vector contains 24 bits with each bit representing the current state of a link in the 4×4 mesh. The link vector of an execution interval is computed by ORing the link usage of all the instructions executed during the instruction interval. This essentially means that, even if a link is used only once during the entire interval, the link vector of the interval denotes that link as being used during the interval. Hence the motivation to have shorter instruction intervals when compared to considerably longer instruction intervals used in other phase characterization works [102] [79].

7.4.2 Runtime Classification

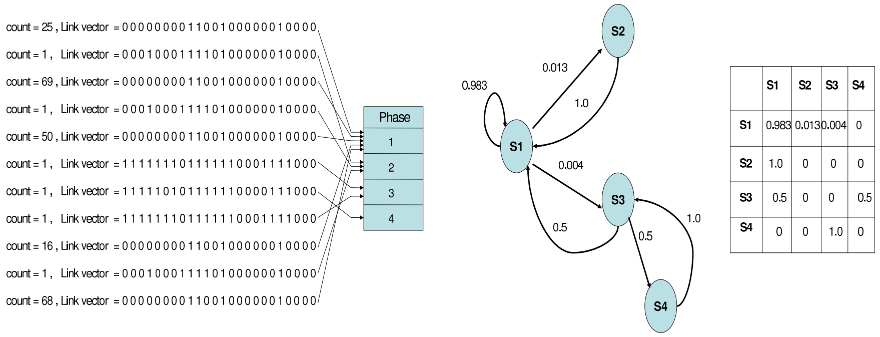
A simple way to identify phases is by using an identifier called "phase id" and a simple way to store phase information is by maintaining a "phase table", with each row containing the link vector which represents the phase and a uniquely assigned phase identifier. A runtime phase classification scheme would thus involve recording all the phases that have been previously encountered in the phase table and (at the end of every new interval) comparing the interval's link vector with the link vectors of the previously-recorded phases (which essentially involves searching the phase table). If there is a match, then that interval is classified as belonging to that phase. If a match is not found, it is a new phase and is added to the phase table with the link vector of the interval and a newly assigned unique phase id. This process can be performed dynamically making it a runtime classification scheme.

7.4.3 Classification Example

Figure 7.3(a) shows a snapshot of the link vectors (of intervals) during a period of execution of the Wupwise benchmark from the SPEC OMP benchmark suite [58]. In this figure, "count", present in each row, indicates the number of contiguous intervals during which the same link vector repeats. The classification (mapping) of intervals to phases which is based on the link vector similarity can be noted.

7.5 Markov Based Prediction

After classifying the intervals into phases as described in the last section, we use a Markov based prediction mechanism to predict the probable link vector of the next interval just before the end of the current interval. Markov based schemes have been used in the past to implement BBV (basic block vector) based phase prediction [103]. This prediction essentially provides the probable link usage information of the next interval. This, in turn can be used to proactively turn off the


count = 25 , Link vector = 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0
count = 1 , Link vector = 0 0 0 1 0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 0
count = 69 , Link vector = 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0
count = 1 , Link vector = 0 0 0 1 0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 0
count = 50 , Link vector = 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0
count = 1 , Link vector = 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0
count = 1 , Link vector = 1 1 1 1 1 0 1 0 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 0
count = 1 , Link vector = 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0
count = 16 , Link vector = 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0
count = 1 , Link vector = 0 0 0 1 0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 0
count = 68 , Link vector = 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0
Phase
1
2
3
4
S1
S2
S3
S4
0.983
0.013
1.0
0.004
0.5
0.5
1.0

	S1	S2	S3	S4
S1	0.983	0.013	0.004	0
S2	1.0	0	0	0
S3	0.5	0	0	0.5
S4	0	0	1.0	0

(a) (b)

Figure 7.3. reffig:linkvecs shows a snapshot of link vectors of intervals during a period of execution of the Wupwise multi-threaded benchmark and the phases they map to. Mapping is done based on link vector similarity. reffig:markov depicts a Markov based transition graph and the corresponding prediction table. Prediction is made based on the probabilities contained in the prediction table. The transition graph shows the transition probabilities pictorially.

links which are predicted to be not used and pre-activate links that are predicted to be used. This pre-activation is done just ahead of time so that the activation latency is hidden and the link is ready for use when the next interval begins. If the prediction turns out to be correct, we stand to save power. However, if the prediction turns out to be wrong, there is a two-fold penalty. First, there is the performance penalty in waiting for the correct links to power on which had been turned off because of the misprediction. Secondly, there is also the power penalty in turning off and then turning on additional links. Therefore, prediction accuracies are crucial to the effectiveness of this scheme. We describe two prediction schemes based on the Markov model in the next two subsections.

7.5.1 Basic Markov Prediction

Markov model is a prediction model used frequently in various domains [104] [105]. A specification of the Markov model contains a set of states and a table, containing the transition probabilities from each state to every other state and itself. With this specification, Markov model can make a prediction about the next state, given the present state. This prediction is based on the transition probabilities. The transition probabilities are continuously built and updated as and when state transitions happen, and therefore, these transition probabilities, at any instant, are based on the previous transition history. A basic Markov prediction involves considering the present state and searching the transition probabilities from this present state to every state and choosing the transition which has the maximum probability. In our context, a state is nothing but the link vector of an interval. Figure 7.3(b) illustrates an example of this scheme. It shows Markov based transition probabilities in the form of a graph and a prediction table at the end of the execution chunk shown in Figure 7.3(a). Each state in Figure 7.3(b) corresponds to a phase in the phase table of Figure 7.3(a). The state S1 corresponds to phase 1, S2 to phase 2 and so on. As an example of Markov based prediction, if the current state is S1, the next state is predicted to be S1 again. As another example, if the current state is S2, then the next predicted state is S1. As a simple illustration of the way transition probabilities are continuously updated, if suppose, S4 now transitions to S2, the new transition probabilities from S4 to S1 and S4 to S4 still remain 0, but the transition probability from S4 to S2 changes from 0 to 0.5 and, the transition probability from S4 to S3 reduces from 1 to 0.5.

7.5.2 Markov Prediction Using a Threshold

This is similar to the basic Markov prediction scheme explained above with one added quality. Instead of making a prediction based on the maximum probability alone, we base the prediction on another parameter called the "threshold". Specifically, we pick the maximum probability prediction and then, check if its probability is greater than or equal to the pre-specified threshold parameter, and if so, we continue as before by choosing the maximum probability next state as our prediction. However, if the maximum probability is less than the specified threshold value, we do not make any prediction. This scheme is intended to weed out predictions which are based on insufficient previous data or are just too close to call. Note that employing a threshold value, in general, decreases the number of mispredictions, as we show later in the results section. For example, in Figure 7.3(b), if the present state is S3, the previous scheme would have predicted either state S1 or S4 to be the next state. In contrast, the threshold based scheme with a threshold of

0.67 makes no prediction (for the present state S3) since the maximum probability entry in the row is less than the pre-specified threshold value. The threshold value is a configurable parameter and can be set high if very little performance impact is tolerated and can be set low if some performance impact can be tolerated with a possibility of higher energy savings.

7.6 Evaluation

7.6.1 Setup

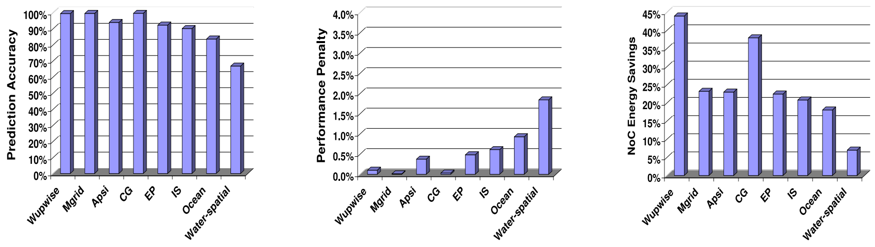
As mentioned previously, we use a 4×4 mesh NoC based 16-core CMP in our experiments. We assume a traditional X-Y routing policy in the NoC. The shared L2 cache is 16 banked SNUCA (static non-uniform cache access) architecture with a bank in every node and each bank is 2MB in size. The link power model we use is taken from [106], and in this model, when a link is turned on, it consumes the same power irrespective of whether it is transmitting data or not due to the link signaling methodology. When a link is turned off, we assume it does not consume any power as in [106]. The power values in the table are obtained from [107]. We use Simics [57] which is full-system simulator combined with a module we implemented to simulate a 4×4 mesh. This setup is used to compute link usage, support routing, and evaluate link power management.

7.6.2 Results

7.6.2.1 Basic Markov Prediction

Figure 7.4(a) shows the link vector prediction accuracy achieved by this scheme for various applications. The main observation is the variation in the prediction accuracies across applications. As can be clearly seen, most applications have prediction accuracies of well over 95%, with Wupwise, Mgrid and CG having accuracies over 99%. Compared to this, water-spatial has a slightly lower prediction accuracy, probably due to the relatively shorter execution time, which in turn results in smaller learning phases.

Figure 7.4(b) shows the performance penalties incurred for different applica-



(a) Prediction accuracy (b) Performance penalty (c) Energy savings

Figure 7.4. Prediction accuracy, performance penalty and the resulting energy savings when the basic Markov prediction scheme is used.

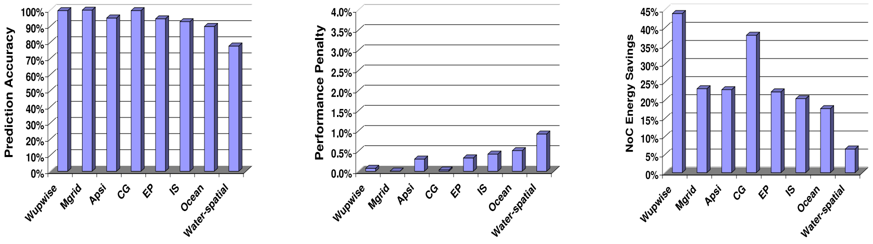
tions, over the case where no link power management is employed. This metric is a reflection of the prediction accuracy. The reason for the observed low penalties is two-fold. The main reason is of course the very high link prediction accuracy. Another reason is the fact that the links that are predicted to be used are turned on ahead of time so that the turn-on latency is hidden and the links are up by the time they are going to be used. The main triumph card of our scheme is the extremely low performance penalties which virtually leaves the original performance unaltered. This is in contrast to other hardware schemes which in many cases incur penalties as high as 12%, as mentioned in [108]. In contrast, our scheme results in penalties below 0.5% in most cases except for water-spatial application, which incurs a penalty of 1.5%. As we demonstrate later, the penalties can be further reduced to being almost negligible.

Finally, Figure 7.4(c) shows the link energy savings achieved by this scheme and as can be seen, Wupwise and CG achieve savings as high as 44% in communication energy.

7.6.2.2 Markov Prediction Using a Threshold

Figure 7.5(a), Figure 7.5(b) and Figure 7.5(c) show the prediction accuracy, performance penalty and the energy savings, respectively, resulting from this scheme with a pre-specified threshold of 0.5. Later, we also present the results with a different threshold value.

The key thing to note is the fact that the performance penalty is further reduced as can be seen in Figure 7.5(b) and yet the energy savings remain almost the same



(a) Prediction accuracy (b) Performance penalty (c) Energy savings

Figure 7.5. Prediction accuracy, performance penalty and the resulting energy savings in the case of Markov prediction using a threshold.

as in the basic Markov prediction scheme. Hence, incorporating a pre-specified threshold results in further fine tuning of the performance penalties. This happens since the threshold parameter filters out predictions which do not have a good prediction history. Employing this scheme results in performance penalty of less than 1% in all cases and less than 0.5% in all but one application.

Chapter 8

Conclusion and Future Work

Technology scaling has had a disparate impact. While, on-chip core counts are increasing at a fast rate, the memory resources are scaling at a much slower rate, and consequently, remain costly and precious. The memory hierarchy resources, such as the on-chip cache and the off-chip memory, play a key role in determing both the overall system throughput and the individual application performance. In order to be used efficiently, the memory resources are generally shared across multiple cores. As a result of being shared, multiple applications/threads can contend and interfere with one another while accessing these resources. This inter-application interference, if not handled aptly, can lead to extensive performance degradation.

This disseration addressed the problem of shared memory resource contention in emerging multicore systems. We studied the causes and different ways in which applications/threads interfere with one another in two key memory resources, namely, the last-level cache and the off-chip memory. We then studied, in detail, the effects of this interference on both system throughput and individual performance. Using these studies, we presented multiple schemes to address the problem of interference and manage the memory resource efficiently. Our approach managed the resource in an application aware manner. Specifically, this involved considering the memory access characteristics of all contending applications and allocating/partitioning the resources based on these application characteristics. We considered both interference reduction and the memory demands of applications while partitioning a resource. The memory resource management schemes presented in this disseration have considered not just resource partitioning and allocation as solutions but also application mapping as a way to reduce interference.

Finally, this dissertation showed that our proposed schemes can be very effective in mitigating the negetive effects of interference on performance. Our proposed schemes improved both system throughput and individual application performance by mitigating inter-application interference.

As core counts continue to increase, emerging multicore systems will have ever higher parallel computational capability. This will enable these emerging systems to solve bigger problems of the future. However, the problem of inter-application/thread interference can pose a significant impediment in future applications extracting the maximum possible parallel computation capability from emergin multicore systems. This dissertation took a significant step in solving this significant problem.

8.1 Future Work

Current application aware state-of-the-art memory scheduling schemes tackle specific problems in naive, application unaware scheduling policies. In an extensive study of all the previous scheduling policies on a large number of workloads in different memory intensity categories, we found that no memory scheduling policy performs well across all the memory intensity categories. For instance, ATLAS [7] and TCM [8] perform well when the memory intensity of the workload is less than or equal to 50%. At higher memory intensities, the naive FRFCFS [22, 23] outperforms both ATLAS and TCM. As part of the future work, we plan to explore a customizable scheduling policy that adapts to the workload charcteristic, so that, performance across all memory intensity categories is better than a fixed memory scheduling policy.

Bibliography

[1] Awasthi, M., D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis (2010) "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques.*

[2] Ramanathan, R. (2006) "Intel Multi-Core Processors : Making the Move to Quad-Core and Beyond," in *Intel White paper, Intel Corporation.*

[3] Hetherington, R. (2005) in *The UltraSparc T1 processor*, SUN.

[4] Tilera *TILE-Gx Processors Family*, http://www.tilera.com/products /TILE-Gx.php.

[5] "http://www.intel.com/p/en_US/products/server/processor/xeon7000? iid=servproc+body_xeon7400subtitle," .

[6] "http://www.dell.com/us/en/enterprise/servers/server-poweredge-r900/pd.aspx?refid=server-poweredge-r900&cs=555&s=biz," .

[7] Kim, Y., D. Han, O. Mutlu, and M. Harchol-Balter (2010) "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA '10: Proceedings of the 15th International Symposium of High-Performance Computer Architecture.*

[8] Kim, Y. et al. in *MICRO '10: Proceedings of the 43th International Symposium on Microarchitecture.*

[9] Muralidhara, S. et al. (2011) "Reducing Memory Interference in Multi-Core Systems via Application-Aware Memory Channel Partitioning," in *CSE Technical Report 11-006, Pennsylvania State University, June.*

[10] Muralidhara, S., L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda (2011) "Reducing Memory Interference in Multi-Core Systems via Application-Aware Memory Channel Partitioning," in *Safari Technical Report TR-SAFARI-2011-002, Carnegie Mellon University, June.*

[11] Muralidhara, S. et al. (2010) "Intra-application shared cache partitioning for multithreaded applications," in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

[12] Muralidhara, S., M. Kandemir, and P. Raghavan (2010) "Intra-application shared cache partitioning," in *IPDPS '10: Proceedings of the 24th international symposium on parallel and distributed systems.*

[13] Muralidhara, S. and M. Kandemir (2011) "Coordinated, Bandwidth Constrained HW/SW prefetching Scheme for Multicores," in *EuroPar '11: Proceedings of the 17th International Euro-Par Conference.*

[14] Muralidhara, S. et al. (2009) "Communication Based Proactive Link Power Management," in *HiPEAC '09: Proceedings of high performance embedded architectures and compilers.*

[15] Mattson, T. G. and G. Henry (1998) in *An overview of the Intel TFLOPS Supercomputer*, Intel.

[16] (2006) "Cell BroadBand engine - white paper," IBM.

[17] Kim, S., D. Chandra, and Y. Solihin (2004) "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.*

[18] Lin, J., Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan (2008) "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," in *HPCA '08: Proceeding of the 14th International Symposium of High Performance Computer Architecture.*

[19] Chang, J. and G. S. Sohi (2007) "Cooperative cache partitioning for chip multiprocessors," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing.*

[20] Vinodh, C., B. Jacob, B. Davis, and T. Mudge (1999) "A Performance Comparison of Contemporary DRAM Architectures," .

[21] Mutlu, O. and T. Moscibroda (2007) "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.*

[22] Rixner, S., W. Dally, U. Kapasi, P. Mattson, and J. Owens (2000) "Memory access scheduling," in *ISCA '00: Proceedings of the 27th International Symposium on Computer Architecture.*

[23] Zuravleff, W. and T. Robinson (1997) "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," .

[24] Dally, W. and B. Towles (2003) *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[25] Duato, J., S. Yalamanchili, and N. Lionel (2002) *Interconnection Networks: An Engineering Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[26] Galles, M. (1997) "Spider: A High-Speed Network Interconnect," *IEEE Micro*, **17**(1), pp. 34–39.

[27] Moscibroda, T. and O. Mutlu (2007) "Memory performance attacks: Denial of memory service in multi-core systems," in *SS '07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium.*

[28] Mutlu, O. et al. (2008) "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture.*

[29] Nesbit, K. J., N. Aggarwal, J. Laudon, and J. E. Smith (2006) "Fair Queuing Memory Systems," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture.*

[30] Rafique, N., W. Lim, and M. Thottethodi (2007) "Effective Management of DRAM Bandwidth in Multicore Processors," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques.*

[31] Yuan, G. L. et al. (2009) "Complexity effective memory access scheduling for many-core accelerator architectures," in *MICRO '09: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture.*

[32] (2010), "The AMD processor roadmap for industry standard servers," .

[33] Casazza, J. (2009) "First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)," in *Intel White Paper.*

[34] *SPEC CPU2006*, http://www.spec.org/spec2006.

[35] Ebrahimi, E., C. J. Lee, O. Mutlu, and Y. N. Patt (2010) "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems.*

[36] Das, R. et al. (2011) "Application-to-Core Mapping Policies to Reduce Interference in On-Chip Networks," in *SAFARI Technical Report No. 2011-001.*

[37] Hur, I. and C. Lin (2004) "Adaptive History-Based Memory Schedulers," in *MICRO '04: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.*

[38] Natarajan, C., B. Christenson, and F. Briggs (2004) "A study of performance impact of memory controller features in multi-processor server environment," in *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture.*

[39] Zhuravlev, S., S. Blagodurov, and A. Fedorova (2010) "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems.*

[40] Snavely, A. and D. Tullsen (2000) "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *SIGPLAN Not.*, **35**, pp. 234–244.
URL `http://doi.acm.org/10.1145/356989.357011`

[41] Chandra, R., S. Devine, B. Verghese, A. Gupta, and M. Rosenblum (1994) "Scheduling and page migration for multiprocessor compute servers," in *ASPLOS '94: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems.*

[42] Verghese, B., S. Devine, A. Gupta, and M. Rosenblum (1996) "Operating system support for improving data locality on CC-NUMA compute servers," in *ASPLOS '96: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems.*

[43] Sudan, K., N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis (2010) "Micro-pages: increasing DRAM efficiency with locality-aware data placement," in *ASPLOS '10: Proceedings of the fifteenth-edition of ASPLOS on Architectural support for programming languages and operating systems.*

[44] Lebeck, A., X. Fan, H. Zeng, and C. Ellis (2000) "Power aware page allocation," in *ASPLOS '00: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems.*

[45] Phadke, S. and S. Narayanasamy (2011) "MLP aware heterogeneous memory system," in *DATE '11: Design, Automation and Test in Europe Conference and Exhibition (DATE).*

[46] Srikantaiah, S., M. Kandemir, and M. J. Irwin (2008) "Adaptive set pinning: managing shared caches in chip multiprocessors," .

[47] component, M. G. D. S. "MT47H128M8HQ-25. http://download.micron.com/pdf/datasheets," .
URL `/dram/ddr2/1GbDDr2.pdf`

[48] Wang, D., B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob (2005) "Dramsim: a memory system simulator." in *SIGARCH Comput. News. 33(4):100-107,.*

[49] Vandierendonck, H. and A. Seznec (2011) "Fairness Metrics for Multi-Threaded Processors," *IEEE Computer Architecture Letters*, **10**, pp. 4–7.

[50] Chandra, D., F. Guo, S. Kim, and Y. Solihin (2005) "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture.*

[51] Snavely, A. and D. M. Tullsen (2000) "Symbiotic jobscheduling for a simultaneous multithreaded processor," *SIGARCH Comput. Archit. News*, **28**(5), pp. 234–244.

[52] Snavely, A., D. M. Tullsen, and G. Voelker (2002) "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," *SIGMETRICS Perform. Eval. Rev.*, **30**(1), pp. 66–76.

[53] (2005) "Platform 2015: Intel Processor and Platform Evolution for the Next Decade," *Intel White Paper, www.intel.com/go/platform2015.*

[54] Song, F., S. Moore, and J. Dongarra (2007) "L2 Cache Modeling for Scientific Applications on Chip Multi-Processors," in *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing.*

[55] Xie, Y. and G. Loh (2008) "Dynamic classification of program memory behaviors in CMPs," *CMP-MSI (in conjunction with ISCA).*

[56] Jiang, Y., X. Shen, J. Chen, and R. Tripathi (2008) "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques.*

[57] Magnusson, P. S. and et al. (2002) "Simics : A full system simulation platform," in *Computer, 35(2):50-58.*

[58] "http://www.spec.org/omp/," .

[59] Cascaval, C. and D. A. Padua (2003) "Estimating cache misses and locality using stack distances," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing.*

[60] Almasi, G., G. A. Asi, C. Cascaval, and D. A. Padua (2001), "Calculating Stack Distances Efficiently," .

[61] Zhang, M. and K. Asanovic (2005) "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture.*

[62] Chang, J. and G. S. Sohi (2006) "Cooperative Caching for Chip Multiprocessors," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture.*

[63] Suh, G. E., S. Devadas, and L. Rudolph (2002) "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," in *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture.*

[64] Suh, G. E., L. Rudolph, and S. Devadas (2001) "Dynamic Cache Partitioning for Simultaneous Multithreading Systems," in *IASTED '01: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems.*

[65] Wu, C.-J. and M. Martonosi (2011) "Characterization and dynamic mitigation of intra-application cache interference," in *ISPASS '11: International Symposium on Performance Analysis of Systems and Software.*

[66] Fedorova, A. (2005) "CASC: A Cache-Aware Scheduler For Multithreaded Chip Multiprocessors." in *Sun Technical Report.*

[67] Jette, M. A. (1997) "Performance characteristics of gang scheduling in multiprogrammed environments," in *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing.*

[68] Devuyst, M., R. Kumar, and D. M. Tullsen (2006) "Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors," in *IPDPS '06: In 20th International Parallel and Distributed Processing Symposium.*

[69] Fedorova, A., M. Seltzer, and M. D. Smith (2007) "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques.*

[70] Azimi, R., D. K. Tam, L. Soares, and M. Stumm (2009) "Enhancing operating system support for multicore processors by using hardware performance monitoring," *SIGOPS Oper. Syst. Rev.*, **43**(2), pp. 56–65.

[71] Tam, D., R. Azimi, and M. Stumm (2007) "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007.*

[72] Chen, S., P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson (2007) "Scheduling threads for constructive cache sharing on CMPs," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures.*

[73] Zhang, E. Z., Y. Jiang, and X. Shen (2010) "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming.*

[74] Beyls, K. and E. H. D'Hollander (2001) "Reuse Distance as a Metric for Cache Behavior," in *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pp. 617–662.

[75] Schuff, D., B. Parsons, and V. Pai (2009) "Multicore aware reuse distance analysis," in *Purdue Technical Report.*

[76] Suh, G. E., L. Rudolph, and S. Devadas (2004) "Dynamic Partitioning of Shared Cache Memory," *J. Supercomput.*, **28**(1), pp. 7–26.

[77] Jaleel, A., W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer (2008) "Adaptive insertion policies for managing shared caches," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques.*

[78] Qureshi, M. K., A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer (2007) "Adaptive insertion policies for high performance caching," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture.*

[79] Sherwood, T., E. Perelman, G. Hamerly, S. Sair, and B. Calder (2003) "Discovering and Exploiting Program Phases," *IEEE Micro*, **23**(6), pp. 84–93.

[80] Watson, D. F. (1994) "Contouring: A guide to the analysis and display of spatial data," .

[81] "http://www.nas.nasa.gov/Resources/Software/npb.html," .

[82] Baer, J.-L. and T.-F. Chen (1991) "An effective on-chip preloading scheme to reduce data access penalty," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing.*

[83] Charney, M. J. and T. R. Puzak (1997) "Profetching and memory system behavior of the SPEC95 benchmark suite," *IBM J. Res. Dev.*, **41**(3), pp. 265–286.

[84] Dahlgren, F., M. Dubois, and P. Stenstrom (1993) "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," in *ICPP '93: Proceedings of the International Conference on Parallel Processing.*

[85] Dahlgren, F., M. Dubois, and P. Stenström (1995) "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, **6**(7), pp. 733–746.

[86] Mowry, T. C., M. S. Lam, and A. Gupta (1992) "Design and evaluation of a compiler algorithm for prefetching," in *ASPLOS '92: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems.*

[87] Mowry, T. and A. Gupta (1991) "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, **12**, pp. 87–106.

[88] Srinath, S., O. Mutlu, H. Kim, and Y. N. Patt (2007) "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA '07: Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture.*

[89] Ebrahimi, E., O. Mutlu, C. J. Lee, and Y. N. Patt (2009) "Coordinated control of multiple prefetchers in multi-core systems," in *MICRO '09: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*.

[90] Tendler, J. M., J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy (2002) "POWER4 system microarchitecture," *IBM Journal of Research and Development*, **46**(1), pp. 5–26.

[91] Palacharla, S. and R. E. Kessler (1994) "Evaluating stream buffers as a secondary cache replacement," in *ISCA '94: Proceedings of the 21st annual international symposium on Computer architecture*.

[92] Vanderwiel, S. P. and D. J. Lilja (2000) "Data prefetch mechanisms," *ACM Comput. Surv.*, **32**, pp. 174–199.
URL `http://doi.acm.org/10.1145/358923.358939`

[93] *Micron: 1GB DDR2 SDRAM component: MT47H128M8HQ-25.*, http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDr2.pdf.

[94] Lee, C. J., O. Mutlu, V. Narasiman, and Y. N. Patt (2008) "Prefetch-Aware DRAM Controllers," in *MICRO '08: Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*.

[95] Jouppi, N. P. (1990) "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*.

[96] Hur, I. and C. Lin (2006) "Memory Prefetching Using Adaptive Stream Detection," in *MICRO '06: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*.

[97] Ipek, E., O. Mutlu, J. F. Martínez, and R. Caruana (2008) "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*.

[98] Liu, F., X. Jian, and Y. Solihin (2010) "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance," in *HPCA '10: Proceedings of the 15th International Symposium of High-Performance Computer Architecture*.

[99] Ebrahimi, E., O. Mutlu, and Y. N. Patt (2009) "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems." in *HPCA '09: Proceedings of the 14th International Symposium of High-Performance Computer Architecture*.

[100] Kim, C., D. Burger, and S. W. Keckler (2002) "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *SIGARCH Comput. Archit. News*, **30**(5), pp. 211–222.

[101] et al., S. C. W. (1995) "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*.

[102] Isci, C. and M. Martonosi (2006) "Phase characterization for power: evaluating control-flow-based and event-counter-based techniques," *High-Performance Computer Architecture, International Symposium on*, **0**, pp. 121–132.

[103] Lau, J., S. Schoenmackers, and B. Calder (2005) "Transition Phase Classification and Prediction," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.

[104] Latouche, G. and V. Ramaswami "Introduction to matrix analytic methods in stochastic modeling," *ASA SIAM, 1999, PH Distributions*.

[105] Joseph, D. and D. Grunwald (1997) "Prefetching using Markov predictors," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*.

[106] Soteriou, V. and L.-S. Peh (2004) "Design-Space Exploration of Power-Aware On/Off Interconnection Networks," in *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*.

[107] Chen, X. and L.-S. Peh (2003) "Leakage power modeling and optimization in interconnection networks," in *ISLPED '03: Proceedings of the international symposium on Low power electronics and design*.

[108] Li, F., G. Chen, M. Kandemir, and M. J. Irwin (2005) "Compiler-directed proactive power management for networks," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*.

Vita

Sai Prashanth Muralidhara

Sai Prashanth Muralidhara joined the PhD program in the department of Computer Science and Engineering at Penn State University in August 2006. He received his B.S. (2004) in computer science from B.M.S. College of Engineering, Bangalore. During his doctoral work, he worked on several research problems pertaining to on-chip caches and off-chip memory. He has published his research in several reputed conferences such as PPoPP, IPDPS, MICRO, HiPEAC, EuroPar and PACT among others.