Techniques to Improve the Performance of Cache Memory for Multi-Core Processors

THESIS

Submitted in partial fulfillment

of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Nitin Chaturvedi

Under the Supervision of

Prof. S. Gurunarayanan

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN) INDIA

2015

Techniques to Improve the Performance of Cache Memory for Multi-Core Processors

THESIS

Submitted in partial fulfillment

of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Nitin Chaturvedi

Under the Supervision of

Prof. S. Gurunarayanan

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN) INDIA

2015

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

PILANI (RAJASTHAN) INDIA

CERTIFICATE

This is to certify that the thesis entitled “**Techniques to Improve the Performance of Cache Memory for Multi-Core Processors**” and submitted by Mr. Nitin Chaturvedi ID No. 2007PHX401P for award of Ph.D. degree of the institute embodies original work done by him under my supervision.

Signature of the Supervisor

Date: (Dr. S. GURUNARAYANAN)
Professor, (Electronics & Instrumentation)
Dean, Work Integrated Learning Programme Division (WILPD), BITS, Pilani

Dedicated

To God

For gifting me with the best

One can have…………….

ACKNOWLEDGEMENTS

This thesis arose in part out of years of research that has been done. During this period, I have met several people who have made a significant contribution in assorted ways to the research and the making of this thesis and who deserve special mention. I am glad to take the opportunity to convey my gratitude to them in my humble acknowledgement.

In the first place, I deem it a great pleasure to express my gratitude whole-heartedly to Prof. S. Gurunarayanan Dean, WILPD, for his supervision, valuable advice, suggestions and guidance from the very early stage of this research as well as giving me extra ordinary experiences throughout the work. Besides providing me unflinching encouragement and support in various ways, he has also allowed me the freedom to experiment with my innovation which has in a major proportion enhanced and nourished my intellectual growth.

I am greatly indebted to Prof. Sudeept Mohan, Prof. Sundar Balasubramaniam and Prof. Abhijit Asati for their crucial contribution, constructive comments and motivation. I thank them for their willingness to share their knowledge with me, which was very fruitful in shaping my ideas and research. Collective and individual acknowledgements are due to all my colleagues who have directly helped me in my work.

Thanks are due to Prof. B. N. Jain, Vice-chancellor and Prof. A. K. Sarkar, Director, BITS, Pilani for the constant support and concern. I would like to gratefully acknowledge Prof. S. K. Verma, Dean, RCD and many others for their indispensable help and for creating a pleasant working atmosphere. I am also indebted to all of my project students at the bits pilani: Arun Subharamanian, Pradeep harinderan, Jithin Thomas, Pranav Gaur, Chirag Aggarwal, Ishan, Kapil, Prashant Gupta.

Words fail me to express my gratitude to my family and friends who were always ready to lend a hand. I thank everybody who was important to the successful realization of this thesis, as well as express my apology that I could not mention personally one by one. Finally, I would like to thank God for always guiding me.

ABSTRACT

Performance gap between the speed of Processor and memory is continuously increasing with advent of every new technology. Compared to traditional super-scalars, Chip Multi-processors (CMP) deliver higher performance at lower power for thread-parallel workloads. However, CMP have further increased the demand for higher on-chip cache capacity as well as off-chip bandwidth due to coherence and capacity-related misses, so there is always a need to judiciously utilize on chip cache memory. This thesis addresses the issues of on-chip shared L2 cache management in the Multi-Core Processors. Now onwards, the last level cache is referred as L2 cache (level 2 cache).

In this thesis, we consider CMP, a class of processors where multiple cores are integrated on to the same chip and each core compete for the total on-chip L2 cache. Two basic schemes are currently used to manage L2 cache. First, a separate cache slice is used as a private L2 cache for each core on CMP. Private L2 caches provide the lowest hit latency but reduce the total effective cache capacity because each core creates a local copy of any block it touches. Second, all cache slices are aggregated to form a single large L2 cache, shared by all the cores. A shared L2 cache increases the effective cache capacity for shared data, but it presents several challenges in the design of an on-chip cache that is shared among multiple cores in CMPs. Our efforts in this work have focused on addressing some of these key challenges.

First, we present a comparative understanding of cache misses in the context of CMPs with shared L2 cache by analyzing the interactions between cache references made by different cores. Then, we propose a novel cache management scheme called adaptive block pinning to mitigate the effect of dominated ownership of blocks within a set by few cores.

Secondly, we focus on one of the most important issues in designing large shared L2 cache in a CMP system which is the increasing dominance of wire delays, which affects the access time and impacts the system performance. In this context, non-uniform cache architectures (NUCA) have proved to be able to tolerate wire delay effect while maintaining a huge on-chip storage capacity. However, the fixed location of data block in NUCA imposes serious limitations with this architecture. In order to overcome this limitation, we propose selective block replication scheme which improve upon the conventional large shared uniform cache and over various NUCA schemes proposed so far, such as Static-NUCA (S-NUCA).

Third, we present solutions for the challenges introduced by dynamic features provided by Dynamic NUCA (DNUCA), like multiple locations for data placement, migration movements and data access policy. To address these challenges we have proposed an adaptive migration-replication (AMR) scheme to overcome the above challenges and reduce miss latency in the NUCA cache along with an efficient data access policy to reduce network traffic.

Finally, we have observed that different applications requires different working set sizes and having varying spatial and temporal localities. Therefore, the performance benefits that can be obtained from fixed configuration caches are limited. Moreover many applications exhibit low spatial locality with few cache words utilized before eviction. This effectively increases miss rate and wastes on-chip network bandwidth. Unused word transfers also consume a large fraction of the on-chip energy. To address these issues, we propose an efficient variable granularity cache design that is tuned to meet the varying runtime locality requirements of different applications.

We evaluated various schemes using full-system simulation using multi-thread, and multi-programmed workloads running on an eight-core CMP. We show that all the proposed shared cache management schemes achieve significant performance improvement over the reference schemes for these workloads. This thesis investigates the problem of sharing of last level cache between concurrently running applications and evaluates cache management schemes as a mean of optimizing the overall system performance. All the proposed cache architectures were simulated and evaluated for performance through simulation studies using Parsec and SPEC 2006 Benchmarks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

CMP	Chip Multiprocessors
UCA	Uniform Cache Architecture
NUCA	Non-Uniform Cache Architecture
SNUCA	Static Non-Uniform Cache Architecture
DNUCA	Dynamic Non-Uniform Cache Architecture
LLC	Last Level Cache
L1	Level One Cache
L2	Level Two Cache
DIR	Directory
L1D	Level One Data Cache
L1I	Level One Instruction Cache
VLIW	Very Large Instruction Word
TLP	Thread Level Parallelism
ILP	Instruction Level Parallelism
FIFO	First In First Out
LRU	Least Recently Used
LFU	Least Frequently Used
MRU	Most Recently Used
SMP	Symmetric Multiprocessor
NUMA	Non-Uniform Memory Access
3C	Compulsory Capacity Conflict

Chapter 1

Introduction

This chapter introduces the need for chip multiprocessor with on chip cache memory. In addition to that, it describes the on-chip cache configurations for CMP and finally it presents the contributions of this thesis.

Chapter 1

INTRODUCTION

1. Introduction

Advances in VLSI technology [1] over the past two decades has enabled the improvement of VLSI systems performance in two ways. Firstly, increase in system operating frequency due to shrinking of transistor sizes. Secondly, implementing several micro-architectural techniques, like super-scalar, out-of-order issue, on-chip caching and deep pipelines supported by sophisticated branch predictors. Unfortunately, as has been recently noted, the future effectiveness of these approaches is limited due to the emergence of two main constraints. The first constraint is increase in the number of transistors and their switching frequency which leads to an overall increase in power consumption. The second constraint is that, as the feature size is decreased, wire delays do not scale efficiently and become a major design limitation for large integrated circuits. These problems have caused a change in the design paradigm of the microprocessor industry [2] [3] [4] [5] [6]. Figure 1.1 shows that the design focus has shifted to Chip Multi-processors, which integrates multiple uniprocessors on to the same die. Chip Multiprocessors (CMPs) are being developed by all the main vendors [8] [9] [10] [11] [12] [13]. However, the sharing of the on chip resources amongst the cores impose new constraints and create new challenges [7] for designers.

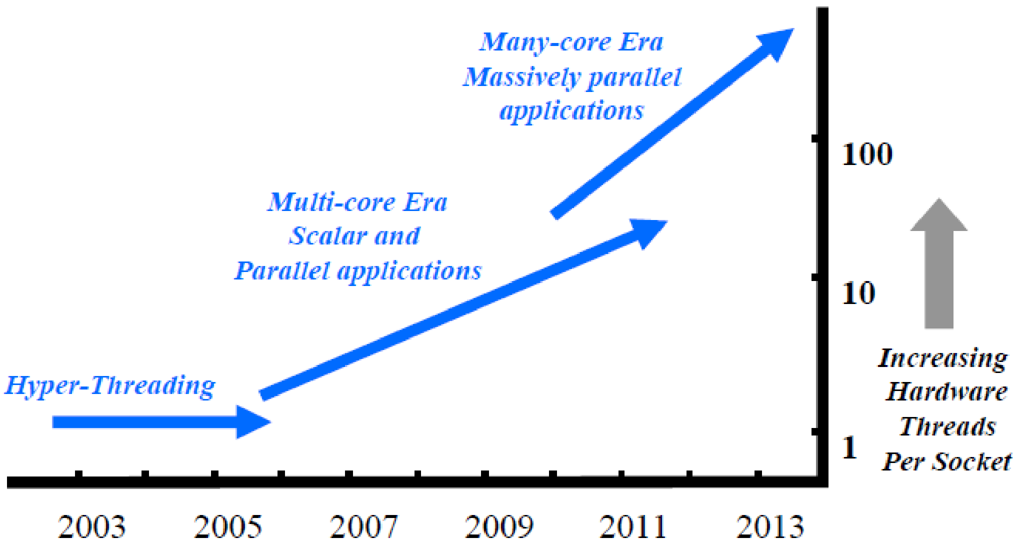


Figure 1.1: Intel processor technology road map, core count increases in next decade

This thesis investigates various design alternatives to improve the performance of the on-chip cache system in CMP architectures. Compared to uniprocessor cache systems, CMP caches have two distinct features that present new challenges. First, the size of the on-chip cache which continue to grow, creating the phenomenon of non-uniform access latency. Non uniform cache architecture allows various parts of the cache to be accessed with different latencies, depending on the physical location. Therefore, a strategic physical placement of cached data can significantly improve performance. Second, the on chip cache system must be able to provide low access latencies to multiple on chip cores simultaneously. Table. 1.1 summarizes the main features of some first generation CMPs from several leading manufacturers. These CMPs show the trend of increasing cache and core count with moderate clock frequencies.

Table 1.1: Comparisons of several leading industry CMPs.

	Year	Cores (Hardware Threads per Core)	Tech. (nm)/ Transistors / Freq. (GHz)	Inter-connect strategy	L2 Cache Configuration size/ assoc / Latency	L2 Cache Sharing Pattern
Server Processors						
IBM Power5	2003	2(2)	130/276M/1.9	Bus	1.9MB/10/13	Shared
AMD Opteron	2004	2(1)	90/233M/2.2	Bus	1MB/16/12	Private
Intel Montectio	2005	2(2)	90/1.7B/1.8	Bus	24MB/12/14	Private
Sun Niagara	2005	8(4)	90/N.A./N.A.	Bus	3MB/8/N.A.	Shared
Embedded Processors						
RMI XLR	2005	8(4)	90/N.A./1.5	Ring	2MB/8/N.A.	Shared
Caviurn Octeom	2005	16(1)	90/N.A./0.6	Bus	1MB/N.A./N.A.	Shared
SiByte BCM 14xx	2005	4(1)	90/N.A./1.2	N.A.	1MB/N.A./N.A.	Shared

The main contributions of this thesis are cache management schemes: an Adaptive Block pinning, Selective block replication and Adaptive Replication-Migration policy for large shared L2 cache along with a proposed novel reconfigurable cache architecture as explained later in section 1.8. These shared cache management techniques achieve significant reductions on cache access latency and communication power over the baseline private and shared designs.

1.1 Need for Chip Multiprocessors

The performance improvement brought by technological advances [14] earlier has slowed down dramatically in past four to five years. This slowdown can be attributing to three key factors as explained below.

First, the most complex micro architectural designs can only bring marginal performance gain at the expense of significantly higher design efforts and longer design cycle. The traditional channels to improve performance by widening the issue widths and using better speculation mechanisms are fundamentally limited by the amount of instruction-level parallelism (IPL). These methods have already reached point of diminishing returns.

Second, higher clock frequencies can no longer be directly translated into better performance because global wire delay does not scale with the silicon feature size. For each subsequent technology generation, less on chip distance can be traversed within one clock cycle, leading to long cross-chip latencies [15] [16]. Thus even though individual chip components continue to become faster, the communication latency among different components remains constant, limiting the performance of the overall system.

Third, power consumption has become a key design constraint that limits achievable processor performance in traditional desktop and server systems. Elevated power density causes transistor reliability and stability problems resulting in higher die temperature. The increasing power usages is the primary reason which finally forces chip designer to deviate evolving traditional super-scalar uniprocessors [3].

1.2 Software Implications

Traditional super-scalar and VLIW architecture exploit instruction level parallelism relying on speculative execution to gain performance. Because the instruction level parallelism that exists in sequential programs is limited even the most elaborate systems today can only achieve a marginal performance gain with better prediction and speculation mechanisms. CMP exploits a much coarser form of parallelism at the thread level which we refer to as thread level parallelism (TLP). For applications with significant TLP, CMP can deliver higher throughput and consume less energy per operation than a wider issue superscalar architecture [17]. Several important classes of application have abundant thread level parallelism and can take advantage of CMP as described below:

1. Server workloads: The large transaction-based server workloads, such as web or data base servers, are inherently thread-parallel because each transaction is an independent task. Today, server workloads are executed on large multichip multiprocessor systems to obtain high throughput. CMPs will work very well for these workloads.

2. Parallel scientific workloads: The Classic algorithms, such as Fourier transform or LU decomposition, are the centerpieces of many critical scientific workloads. Similarly, large compute intensive programs such as weather forecasting demand extremely high performance that uniprocessors are unable to deliver. Because of their importance, they are well studied and heavily parallelized at the thread level to take advantage of large multichip systems. These scientific workloads will work even better on CMPs because they have tighter integration that reduces communication latencies among different cores and memory.

3. Multi-Programmed workloads: Most commercial modern operating systems support multitasking and can run a large number of programs in parallel. In fact, desktop machines today run hundreds of programs concurrently using time-sharing. Thus, we anticipate multi-programmed workloads to be the most common ones for a desktop processor. Multi-programmed workloads are naturally thread parallel as different programs rarely share data, thus fully utilizing the features of a CMP.

1.3 Hardware Implications

From a hardware point of view, CMPs address three key bottlenecks of unicore processors: (1) Power budget, (2) Global wire delays, (3) Design complexity as described below:

1. Power budget: CMPs achieve high performance by running different threads in parallel, putting less pressure on individual thread performance. Thus, CMPs can use relatively less aggressive cores and scale back clock frequency. This approach sacrifices some single-thread performance, but allows many power-inefficient features to be removed from the processor, thereby reducing energy per operation.

2. Global Wire Delay: The physical structure of a CMP naturally constricts the majority of the data movement to be localized within each processor core. Global wires in a CMP will mainly be responsible for transporting shared data between different threads. While increasing global wire delay will remain a problem, such global communication happens much less frequently compared to, for example, access to the register file in a wide super-scalar processor. In addition, this abstraction gives more

control over the wire delay problem to the software. For example, the operating system can place multiple threads that have a high degree of sharing in adjacent cores to minimize the cost of global communication.

3. Design complexity: The CMP approach reduces design complexity by allowing the chip makers to reuse previous core designs with minor modifications to suit future products. The focus of the redesign effort is the interconnection network responsible for communication among cores, caches, physical memory, and I/O devices. Thus CMPs can have a much shorter design cycle and time to market compared to super-scalars (refer Figure 1.2).

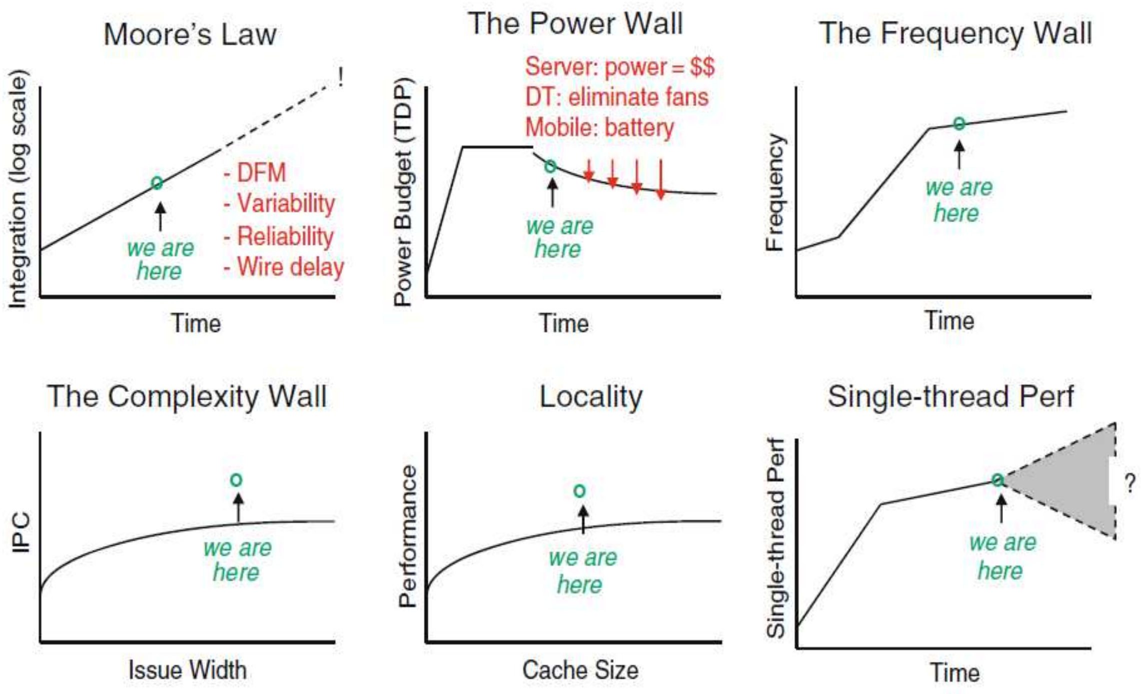


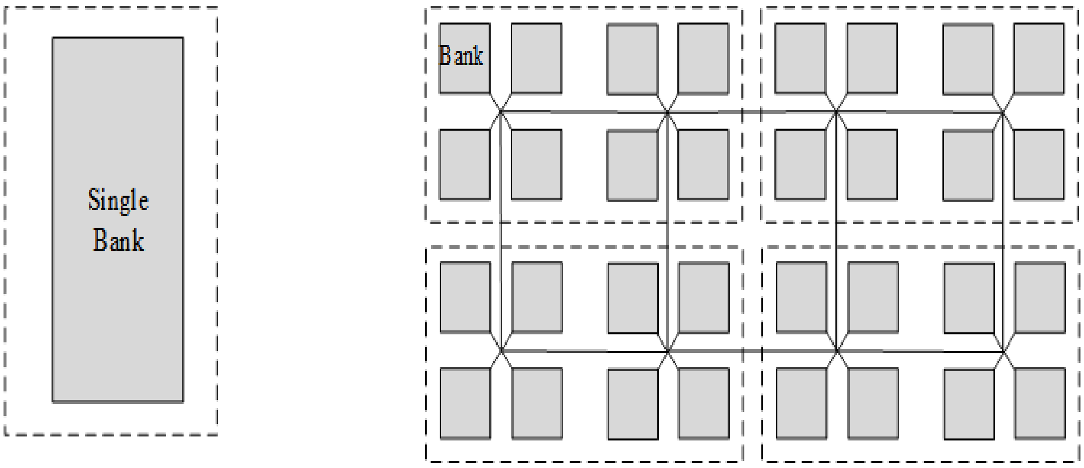
Figure 1.2: Cache design complexity

1.4 CMP Design Trends

There are two trends in future CMP design; First, CMPs will have more cores. For example, the Niagara [18] and XLR chips have 8 cores and cavium octeon CN38xx chip has 16 cores as shown in Table. 1.1. Each core is likely to be relatively simple, especially in the embedded chip space. Second CMP will have more total cache capacity. For example, the newest Intel Montecito chip, based on the Itanium [11], has two cores, each with its own 12 MB L3 cache, forming a total on-chip capacity of over 24MB.

1.5 Non-Uniform Access Latency

Traditional cache architectures are uniform cache architecture (UCA) as shown in Figure 1.3(a) where the access latency to each location is same. Most current cache designs divide large caches into small slices as shown in Figure 1.3(b) to reduce both access latency and energy consumption. The cache access latency is primarily dominated by the access time of each individual cache slice, thus the access latencies to various slices are fixed.



(a) UCA (b) NUCA

Number of banks: 1 bank : 32 banks

Average loaded access time: 255 cycles : 24 cycles

Figure 1.3: Uniform cache access (UCA) Vs Non uniform caches architectures

In the larger caches anticipated in future CMPs, wire delay [16] will cause cross-chip communications to reach tens of cycle. Cache fetch latencies will be dominated by the wire delay to reach each individual cache slice rather than the time spent accessing the slice itself. The access latencies to various slices will become significantly different depending on their location with respect to the load/store unit of the processor. UCA design is no longer suitable for these wire delay dominated caches because using the worst-case latency will result in unacceptable hit times. Thus, we must allow different slices of the cache to be accessed at their fastest possible latencies. The resulting cache design is what we refer to as a non-uniform cache Architecture (NUCA) [19]. Figure 1.4, illustrates this cross-chip latency.

A NUCA architecture can be either static or dynamic. A static NUCA(S-NUCA) simply relaxes a UCA design and allows different cache slices to be accessed with different latencies. It is static because each cache block is still statically mapped to a specific bank.

The more flexible dynamic NUCA (D-NUCA) cache exposes the physical location of each block to the designer, allowing more optimal placement than the statically address-mapped approach of S-NUCA.

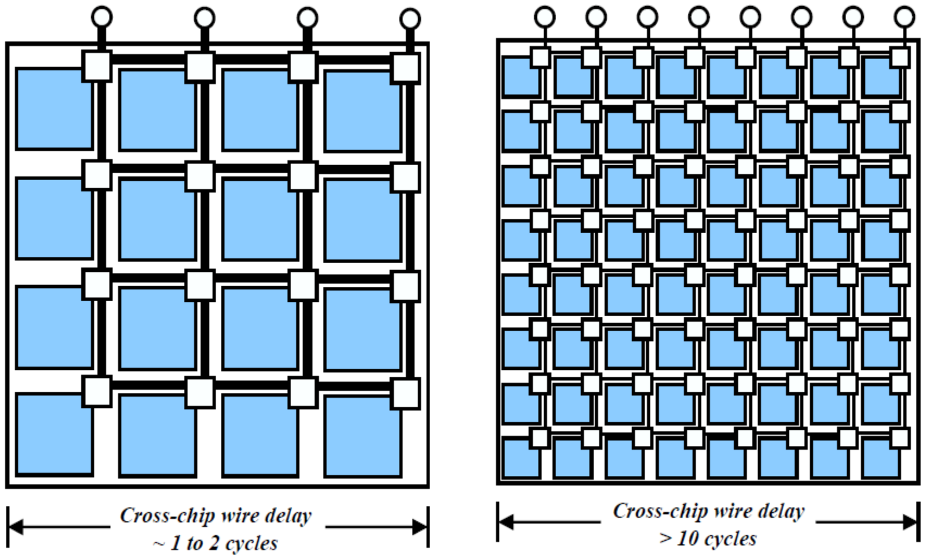


Figure 1.4: Non-uniform cache architecture (NUCA) has long cross-chip latency, wire delay dominates overall cache access time.

The more flexible dynamic NUCA (D-NUCA) cache exposes the physical location of each block to the designer, allowing more optimal placement than the statically address-mapped approach of S-NUCA. An intelligent placement maps the data to physical cache locations so that the working set of the workload stays in the cache slices which are physically closest to the core. Such a placement minimizes the cross-chip communication latency incurred by cache accesses. However, the process of locating a cache block in a D-NUCA can cost significantly more time and energy as compared to S-NUCA**.**

1.6 Thesis Focus: Shared L2 Cache Management

1.6.1 Thesis Problem Statement

For any computer system, its overall performance is often directly correlated to the performance of its memory hierarchy. In CMPs, off-chip misses will remain expensive but increase in clock frequency together with worsening global wire delays will also increase latencies for cross chip communication. Effective use of on-chip cache must therefore consider both the cost of off chip misses and the cost of cross chip communications. Two base-line last level cache designs private and shared illustrate the trade-off between these two components of effective data access latency. For simplicity we assume in the rest of the thesis that the second level cache (L2) is the last level of on chip cache.

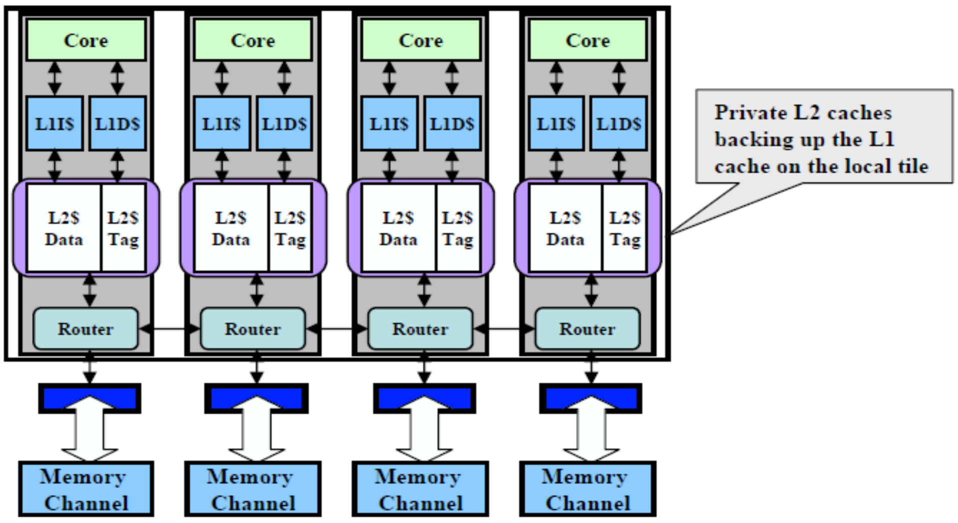


Figure 1.5: Private design of L2 caches

A private design eventually partitions all of the on chip L2 cache slices such that each processor is assigned to its closest partition as its private L2 cache as shown in Figure 1.5. The shared design aggregates all the L2 cache slices to form a single L2 cache slice shared by all the cores as shown in Figure 1.6.

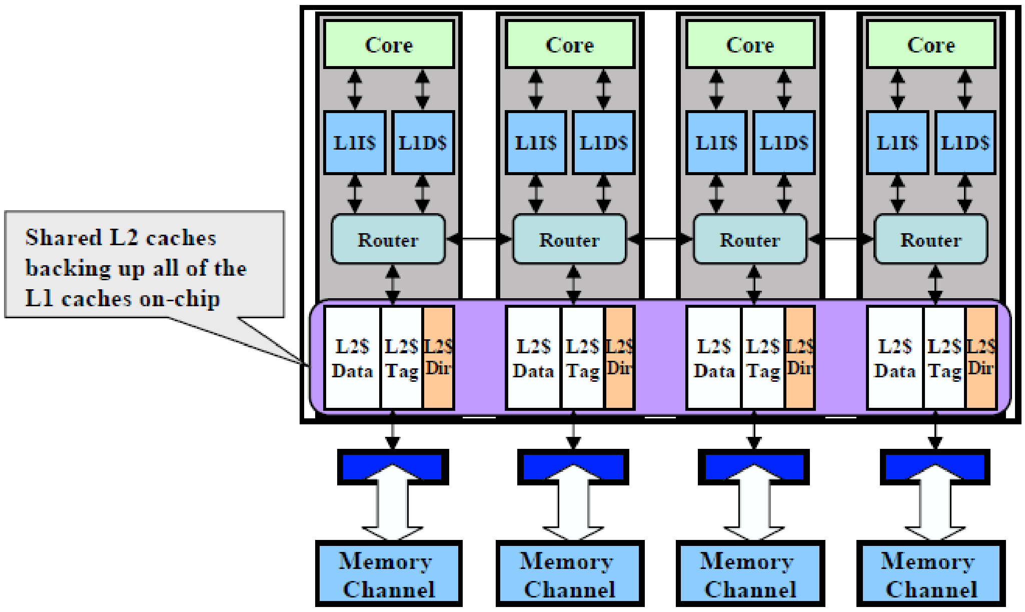


Figure 1.6: Shared design of L2 caches

The private design has a low L2 hit latency as the private L2 cache is physically co-located with the processor core and has a much smaller area than a shared cache. This layout provides good performance if the working set fits with the local L2 slice. The disadvantage of the private design is that effective on chip cache capacity is reduced for shared data as each core must retain its own copy of shared data block. The shared design reduces the off chip miss rate for large shared working sets because only a single on chip cache copy is required for any shared data.

However large shared L2 caches have worse access latency than a small private L2 cache. With multiple cores, this placement task becomes particularly challenging because many cores may contest for the same shared data simultaneously and the optimal placement of the shared data in cache may not be close to any of the requesting cores, thus impacting the access rate. In this thesis, we have investigated various cache management policies for large shared Last Level cache in CMP. We studied private and shared cache designs and explored novel cache management schemes with optimal trade-offs between the off chip miss rate and the cross chip latency to achieve low data access latencies for future CMP.

1.7 Evaluation Metrics for CMP

This section presents the evaluation metric employed in this thesis. We do not focus on traditional uniprocessor metrics such as IPC since it is not the correct metric to evaluate CMP performance.

1.7.1 Latency

CMP running many commercial, scientific, and data-mining workloads exhibit abundant thread-level parallelism, and thus using multiple processors is an attractive approach for increasing their performance. To support the frequent communication and synchronization in these workloads efficiently, servers should optimize the latency of cache-to-cache misses. A cache-to-cache miss is a miss, often caused by accessing shared data that requires another processor’s cache to supply the data. To reduce the latency of cache-to-cache misses, a coherence protocol should ideally support direct cache-to-cache transfer. Our goal in this thesis is to reduce the access latency of shared caches in CMPs. Execution time is the ultimate effect of latency to the system performance and we use that as an evaluation metric.

1.7.2 Network Traffic (Bandwidth)

A cache coherence protocol should conserve bandwidth to reduce power consumption and avoid interconnect contention, because contention reduces performance. Past research has extensively studied the bandwidth efficiency of different cache management schemes and coherence protocols. We use both on-chip and off-chip network traffic (bandwidth) as evaluation metrics.

1.7.3 Effective Cache Utilization

The increasing gap between processor and memory speed and increased number of cores in the system make maximizing on-chip cache capacity crucial to achieving good performance. If the effective on-chip cache capacity is small, the number of off-chip misses will increase, which hurts system performance severely due to increased off-chip bandwidth and corresponding higher energy consumption. In order to measure how effectively we improve utilization we use cache miss rate, (misses per kilo instructions (MPKI)), as our performance metric.

1.7.4 Energy/Power Consumption

With the increased performance and clock rate of processors, processor power consumption and heat dissipation have become one of the challenges in the design of high-performance systems. Monolithic processors have reached a level where they consume large power resulting in less performance improvement per unit power; as a result, industry moved to multi-core on a chip for performance growth while depending less on raw circuit speed and, thus, power. We estimate the dynamic energy in the on-chip memory hierarchy to be roughly 30% of overall chip energy consumption. We use dynamic energy (and hence dynamic power) consumption as the evaluation metric. Although we do not model leakage (static) power for the full system or dynamic power for the cpu logic, we can put the dynamic power of on-chip memory hierarchy into perspective by looking into some prior studies [5].

1.8 Thesis Contribution

The most important contributions of this thesis are:

- **Adaptive block Pinning Technique.** We have proposed and evaluated a hardware-based approach, called block pinning, for eliminating inter-processor misses and reducing intra-processor misses in a shared cache. Furthermore, we showed that an adaptive block pinning scheme provides improvement over the benefits obtained by the block pining and set pinning scheme by significantly reducing the number of off–chip accesses. This work also proposes two different schemes of relinquishing the ownership of a block to avoid domination of ownership of few active cores in multi-core system which results in performance degradation. Extensive analysis of these approaches with SPEC and Parsec benchmarks are performed using a full system simulator.

- **Selective block Replication Scheme.** We proposed and evaluated selective block replication scheme which improve upon the conventional large shared uniform cache and various NUCA schemes proposed so far, such as S-NUCA, SPNUCA in terms of average access latency without significant reduction in the hit rate. This scheme dynamically keeps track of frequency of usage of the remote blocks and selectively replicates the highly used block in the local bank cluster of the requesting cores. The complete set of L2 cache is divided into various bank clusters. Each core has a local bank cluster which is close to it and a central bank cluster. This scheme allows use of both shared as well as replicated blocks. An extensive analysis of our proposed scheme as compared to static NUCA using SPEC and Parsec benchmarks are performed using a full system simulator.
- **Adaptive Replication-Migration Scheme (AMR) with data access policy**. NUCA partitions the complete cache memory into smaller multiple banks and allows banks near the processor core to have lower access latencies than those further away, thus reducing the effects of the cache's internal wire delays. Our proposed AMR scheme uses migration scheme to move blocks close to the requesting core in addition to the selective block replication scheme to keep most frequently used blocks within the local bank cluster of the requesting core and prevent data ping-ping effect. Previous work considered D-NUCA as a promising design. In our work, we proposed an efficient data access algorithm for NUCA design using a set of location pointers with in each bank to reduce miss latency and on-chip network contention. Extensive analysis shows that our proposed AMR scheme along with data access scheme reduces dynamic energy consumed per memory request, and achieves an average performance speedup as compared to S-NUCA and D-NUCA cache management schemes.
- **A novel reconfigurable cache architecture with adaptive block size.** Data movement between cores shared cache and its management impacts memory access latency and power. The efficiency of high-performance shared memory multi-core processors depends on the design of the on chip cache hierarchy and the coherence protocol. Current multi-core cache hierarchies uses a fixed size cache block in the cache organization and in the design of the coherence protocols. The fixed size of block in the set is basically chosen to match average spatial locality requirement across a range of applications, but it also results in wastage of bandwidth because of unnecessary coherence traffic for shared

data. The additional bandwidth has a direct impact on the overall energy consumption. In this work, we present a new adaptable cache design that matches data movements with the spatial locality of the application.

With the contributions described above, we have developed faster and more efficient shared cache management schemes that provides larger effective on-chip cache capacity, faster data availability, reduced L1 miss penalty, reduced last-level cache miss, reduced interconnect and off-chip bandwidth requirement, and reduced dynamic power consumption.

1.9 Organization of Thesis

This thesis is structured in eight chapters.

Chapter 1 highlights the advantages of CMP architectures and the problems that this research attempts to investigate. The remainder of the thesis is arranged as follows.

Chapter 2 reviews the background information related to the traditional cache architecture, first generation CMP cache architecture, and network on chips, cache coherency, cache simulators and shared cache memories. It also reviews several schemes that attempt to improve the efficiency of cache hierarchies both in the single processor and the CMP domain.

Chapter 3 describes the experimental methodology followed in this thesis, we describe the CMP working environment with processor and cache simulator used as well as their integration. It also provides a short overview of the benchmark used for the evaluation of different proposed shared cache management schemes.

Chapter 4 describes a proposed novel adaptive block pinning scheme to manage unwanted block eviction and block relinquishment policy to dynamically relinquish the owner ship of the cache block in shared cache architecture.

Chapter 5 describes the detailed implementation of non-uniform cache architecture for multi-cores and the proposed selective cache line replication scheme for non-uniform cache architectures.

Chapter 6 presents adaptive migration-replication (AMR) scheme which combines the advantage of selective block replication and block migration to reduce both off-chip and cross-chip access latency. We also present the implementation of a novel data access policy to manage network traffic on the chip. These are evaluated together with other schemes that were developed during this work. The advantages and drawbacks of each scheme are identified which is then used to develop a novel shared cache management scheme.

Chapter 7 describes a novel reconfigurable cache architecture that adapts according to the applications executed on the processors.

Chapter 8 concludes this thesis by summarizing the contributions made in addition to future directions and possibilities. Finally, in Appendix A, we give an overview of the cache coherence protocol used in this thesis.

Chapter 2

Background and Literature Review

This chapter discusses the design choices that can be found in literature for cache organization and for the design of cache coherence protocols for multiprocessors. This chapter presents an overview of current cache coherence protocols and discuss several alternatives to design the cache hierarchy in CMP architectures.

Chapter 2

Background and Literature Review

2.1 Introduction

In this chapter, we will briefly discuss background information related to this work. We will begin in section 2.1 by discussing conventional cache design techniques and existing cache replacement policies. In section 2.2 we describe cache hierarchy and cache partitioning for chip multiprocessor along with various existing cache design techniques for multicores to improve system performance. Finally, section 2.3 presents an overview of existing cache coherence protocols developed for current multiprocessors.

2.1.1 Conventional cache architectures

One of the major components of architecture level power consumption is the memory subsystem [21]. Benini et al. [22] analyzed in detail various architectures and optimization techniques used in memories. Panda et al. [23] surveyed various techniques used in memory related optimizations in embedded systems. As per the existing research 42% and 23% of the total processor power in StrongARM 110 [24] and Power PC respectively is used by the cache. According to these numbers, there is substantial influence on the overall energy utilization when cache energy consumption is reduced.

Several hardware (architecture level) and software techniques have been proposed to lower the consumption of power and enhance the memory subsystem performance. Each of these techniques has its own merits and demerits. The hardware techniques may result in intricate circuit implementation while incorporating a variety of applications. The software methods adjusted for a specific application cannot be reemployed for any other applications. These issues are extremely important for system design as increase in the cost of hardware pushes the system towards non-application specific designs.

Lowering the consumption of the cache power can be attained by lowering the number of cache misses, latency (delay) per access, shutting down a part of the cache, reconfiguring the cache for specific applications. Various architecture level techniques described in literature to attain these, include hardware prefetching [27], vertical cache partitioning, horizontal cache partitioning, reconfiguring cache architecture [38], optimizing cache control circuitry, modifying the replacement circuitry to improve hit rate [32] [33], making use of the compiler and operating system information (software controlled cache) to improve performance and various combinations of some of these [40].

While designing a cache, one has to choose between the direct-mapped and set-associative mapping schemes as these are the existing energy proficient mapping techniques [25]. Both these schemes have their own advantages and disadvantages in context of cache access time, dynamic power consumption and cache hit rate. Literature shows that direct-map cache consumes much lesser dynamic power per cache access compared to a set-associative cache. For instance, Hennessy and Patterson reported 55% more dynamic power consumption per access for a 4-way set-associative cache as compared to that of a direct-map cache.

An experiment conducted by Hennessy and Patterson [25] indicated that a rise in associativity results in a lowering of the miss rate and thus, lowered the consumption of power. This indicates that for applications needing a high cache hit rate and low energy consumption it is preferable to use a set-associative cache, despite it the additional cost related to consumption of power due to increase in tag comparison. For instance, for a direct-mapped 8KB cache, the average miss rate for the SPEC92 benchmarks is 4.6% while it is 3.8% for two way and 8KB set-associative cache and 2.9% for a 4-way 8KB set-associative cache. Though the miss rate reduction is small, it results in a significant performance improvement which depends heavily on the hit rate and access time, as the large cycle penalty of a cache miss is now avoided. So, if we measure the performance of a cache in terms of the power consumption, the set associative cache may give better performance than the direct-mapping scheme because energy overhead due to miss penalty is much higher than per access power. Thus, applications that need a higher cache hit rate favor a set-associative cache over a direct-mapped cache. The cache power consumption characteristic varies with the total cache size as well [26]. Small cache size is energy proficient and has less access latency but suffers because of poor hit rate. Set-associative mapping scheme also provides support for energy efficient caching schemes like way shutdown, way concatenation, way prediction and process aware caching efficiently. Thus, set-associative mapping scheme is chosen for this work.

2.1.2 Replacement schemes in caches

The three types of misses incurred in the cache are the compulsory, capacity and conflict misses. A compulsory miss is the result of the first access to a block that has previously never existed in the cache. A capacity miss occurs when the cache is not big enough to accommodate all the blocks required to efficiently execute the program. A conflict miss takes place when multiple blocks map to the same set. This occurs in the direct-mapped and set-associative cache, but not in the fully associative cache. Conflict misses are one of the major cause of cache misses during program execution.

The performance of a cache replacement techniques chiefly relies on how precisely the cache can envisage the future reference pattern depending on previous references. The future reference pattern may depend on the past reference pattern and input data. It is relatively easy to find the reference pattern in a static scheduled system than in a dynamic-scheduled system. The choice of a replacement policy is one of the most crucial cache design problems. Selection of a suitable line/block replacement algorithm, in the case of fully associative and set-associative caches [28], can have significant impact on the overall system performance. The existing processors use different replacement strategies including random, round robin, First-In-First-Out (FIFO), Least Frequently Used (LFU), Least Recently Used (LRU), Pseudo LRU (PLRU), MRU (Most Recently Used) and variants of these [29] [43] [44]. The performance of all these policies are compared and analyzed with reference to the optimal replacement policy (OPT). This strategy cannot be implemented in the instance of dynamic scheduling systems, since the future cache references are not accessible [30]. Even if the future references are known, it is impractical to implement this scheme because of the computational complexity involved in finding the cache line to be evicted. However, it is very useful in determining the lower limit for the number of cache misses.

The least frequently used (LFU) replacement scheme selects the cache line to be evicted based on the frequency of access of the cache lines. LFU requires maintaining a frequency count register per cache line and is incremented by one, each time a reference is made to the cache line. So a register is updated for every cache access. LFU finds the cache line with the lowest frequency count as the one to be evicted. LRU and its variants are the commonly employed replacement policy in the cache on account of their high performance [29]. There are different techniques to implement LRU in hardware, which comprise of Counter, Square matrix, Skewed matrix, link list, Phase, and Systolic array method [31]. The replacement circuit intricacy and the additional hardware needs are comparatively less than the LRU and LFU in the instance of FIFO and Random replacement schemes [34] [35]. A variant of LRU replacement policy is Early Eviction LRU (EELRU) [37]. The EELRU dynamically opts to remove the LRU page or the most recently used page. Maki et al. [39] try to improve the LRU replacement decision with the help of an additional bit (lock/release) per cache line and lock and release operations. This process aware scheme reported 60.9% reduction in cache miss ratio and faster execution then the LRU replacement strategy. Wang et al. proposed a replacement algorithm which improves the cache hit performance or in the worst case performs similar to LRU for set-associative caches [40]. Wong and Baer [32] proposed an enhancement for the LRU replacement policy with a temporal bit per cache line. This temporal bit acts opposite to the EM bits in [40], i.e. it specifies the cache lines to be retained in the cache rather than cache lines to be evicted. The temporal bit settings are determined by off-line profiling or an on-line hardware history table. This bit set when there is a cache hit in that line and is reset when non-LRU line is evicted from the set. Martin Kampe et al. proposed self-correcting LRU [36], which depends on LRU, supplemented with a feedback loop to continuously oversee and revise the mistakes done during replacement. O'Neil et al. proposed the LRU-2 method [41] that evicts the memory block with a minimum time stamp of the second to last reference. A hardware history table is used by Lai et al. [42] to envisage when a cache block is dead and which block to pre-fetch and replace the dead one. The fact that the size of the history table restricts the length of the history consulted is the disadvantage of this algorithm. The technique of merging any two extant replacement strategies is indicated by Yannis Smaragdakis [45]; this technique is extremely suitable with real program data, frequently surpassing LRU (in addition to all the other policies it adopts) by over 40 %. Jaafar Alghazo et al. [46] proposed SF-LRU (Second Chance-Frequency-Least Recently Used) that merges LRU and Least Frequently Used (LFU) using the second chance concept. Outcomes of experiments conducted indicate that the SF-LRU crucially lowers the number of cache misses in contrast to the LRU (up to 6.3%) and LFU (up to 9.3%). SFLRU [36] has been recommended to attempt to partly consider the frequencies while making the LRU decisions and to ensure that the costs are less. Most Recently Used (MRU) policy selects the most recently used cache line from a set for eviction. This algorithm is not widely used in the cache memory system because of its bad temporal locality. In addition to all these replacement policies, there exist various replacement strategies which are very specific to architectures like victim cache [48], skewed-associative cache, elbow cache etc.

2.1.3 Energy efficient cache architectures

Cache memory analysis reveals that the chief sources of power consumption are the data lines and data sense amplifiers. The power consumption by the data lines and data sense amplifiers as per Wilton and Jouppi [26] were 55%, 65% and 75% of the total cache sub system power consumption for the direct-mapped, 2-way set-associative and 4-way set associative mapping schemes respectively. One way to minimize the dynamic power consumption is to mitigate the intrinsic activities of the cache during a cache access. Minimum cache power consumption can be achieved if the cache access incurs minimum conflict misses. Also, if each cache hit results in reading and comparing only one tag entry, then enabling and accessing only that one data entry and if each cache miss results in only reading and comparing one tag entry also helps in reducing power consumption.

Hardware pre-fetching [27] is an accepted method to improve the cache performance in conventional systems. Pre-fetching methods attempt to lower the cache miss rate by pre-fetching instructions into internal cache. This may results in replacing useful data in the cache. Unfortunately, most of the existing pre-fetching methods are not extremely efficient in embedded systems because of real time processing constraints.

Embedded systems employ various partitioning schemes to make cache energy efficient and deterministic. This ensures the smooth execution of higher priority time-critical tasks. A cache partitioning can be either static (fixed) or dynamic. A fixed partitioning scheme partitions the entire cache into N equal/ unequal sizes and assigns them to the tasks. In case of dynamic partitioning scheme, cache is partitioned based on various parameters such as size of the task, priority of the task, number of cache blocks in use etc. Another way of partitioning the cache is vertical and horizontal partitioning.

2.1.4 Operating system support

CMP incorporate novel hardware features that are dissimilar from traditional uniprocessors or conventional symmetric multiprocessing (SMP) multiprocessor systems. These novel features bring additional performance improvement possibilities and problems. Chip multi-processors architectures deal with three basic difficulties in further expanding processor clock frequencies further. To start with, the performance difference between the speed of the processor and memory forces processors to halt for the vast majority of their time for the memory to deliver the information, making recurrence increments in frequency insufficient.

Secondly, vital utilization and high temperature dispersal of processors, which are attached to the frequency of CMP, are reaching their physical breaking points. Lastly, higher frequencies need deep execution pipelines, making the design configuration and verification of advanced processors even more difficult and challenging.

Consequently, Chip multi-processor have turned into the new mainstream architecture [49] and henceforth obliges prime consideration from software programming engineers to have a Chip multi-processor aware operating system and applications framework. Considering the software perspective, the skill to use the maximum capacity of various execution cores in chip multi-processor has turned out to be troublesome, as it includes many software programming layers. At the higher level, each core is used to execute an alternate application, or a solitary application must be parallelized, either naturally or manually, into various threads. On the other hand, when application level parallelism is separated and communicated clearly, there are two difficulties in accomplishing adaptable execution that are present in chip multi-processor architectures.

Contention on Shared Resources: In contrast to the conventional SMP (symmetric multiprocessor) systems, there are more shared resources on the critical path in each individual core in a chip multi-processor. Some of these resources comprise of on-chip shared last level caches, the memory controller, and the interconnection network to other processor sockets (or the I/O fabric). The presence of uncontrolled contention in any of these shared resources may lead to a degraded system throughput and hampers performance.

Non-Uniform Inter-Core Communication Latency:

In contrast to traditional SMP systems, chip multi-processors are hierarchical in nature, and the communication latency between two cores in chip multi-processor varies substantially and relies on their physical closeness. For instance, in contrast to cores that are located on two different chips, same chip cores can interact quicker via on-chip caches. This facet of chip multi-processors is similar to conventional NUMA (non-uniform memory access latency) multiprocessor systems. Effectively, chip multi-processors designs include extra levels to the memory hierarchy and, consequently, result in the non-uniformity of communication latency much more pronounced as compared to conventional NUMA systems. Consequently, handling these issues, to a certain extent, falls under the purview of the operating system. Apart from having knowledge of the basic CMP hardware design, the software system (operating system) can remove and include information related to the dynamic character of the running system, which comprises of how well the hardware, and how well the software applications are performing , thus allowing the operating system to deal with the resources in an efficient way. So as to ensure the same, the operating system precisely recognizes and quantifies the latency inflicting events in an intricate multi-core system. At the operating system level, there have been three crucial methods to deal with the issue pertaining to consumption of energy: process scheduling techniques [20], paging systems [50], and performance tuning [51].

In summary, chip multi-processors addresses the primary issues that are present within the evolution of chip multi-processors like the power wall, memory wall and design intricacy. The basic shift in chip multi-processor design needs in depth support from software's so as to attain the complete potential in terms of processing speed.

2.2 Conventional cache design limitations

The mechanism and policies utilized for conventional caches architectures that support uni-cores have several limitations when they are used with CMPs. Firstly, these schemes are insufficient to handle the competition among the cores for the on chip caches. Secondly, these policies failed to support physical memory sharing between rival threads, and to avoid damaging intrusion like thrashing. Lastly, fairness improvement, QoS guarantee and priority supports are other limitations of conventional caching policies. There are no traditional cache proposals that deal with all CMP caching needs.

2.2.1 Caching for Chip Multiprocessor

CMP is the novel standard for high-performance computing. The chip designers raises the number of processor cores in the chip so as to benefit from the thread-level parallelism and frequency is reduced to lower the consumption of power. A lower frequency not only saves power but also lowers the processor-memory performance gap and thus harmonizes the architecture to some degree.

Although, CMP caching presents a number of new challenges to CMP cache designers, these challenges are not new in the history of general caching analysis & research. It has been dealt with in varied caching systems including virtual memory paging, conventional shared-memory multiprocessor memory designs and web caching. The notion of caching was introduced and recorded within the IBM System/360 implementation [102] that employed a high-speed buffer to reduce the processor-memory speed discrepancy by utilizing the locality principle [40]. The cache's efficacy is to a great extent ascertained by its data placement, access and replacement strategies for a specific cache size. Majority of the linked studies pertaining to CMP caching emphasizes on the memory organization of shared memory multiprocessors. Figure 2.1, indicates a Dual core CMP with multi-level cache and their on-chip access latencies. In this figure, two cores share the last-level cache. The latency for cache hits in the first-level cache is the interaction time for roundtrip to the cache in addition to the hit time for the cache. If there is a hit in the last-level cache, the latency is the (round-trip) interaction time (both between processor and first-level cache, and between first-level and last-level cache), miss time for the first-level cache and hit time for the last-level cache. For misses in all levels of cache, latency comprises of miss times for all caches, (round-trip), interaction time (also comprising of off-chip interaction) and latency of primary memory. The first-level cache characteristically has a latency of 2-3 clock cycles and the ability of around 16-64 Kbytes. On-chip last-level caches in comparison have bigger storage abilities and latency. Latency is characteristically in the extent of 8-30 clock cycles and capacity is in megabytes with the extant high-end processors. The increased latency of the larger memory blocks is the result of the distance to the memory block on account of the bigger size and the look up time in the bigger memory block.

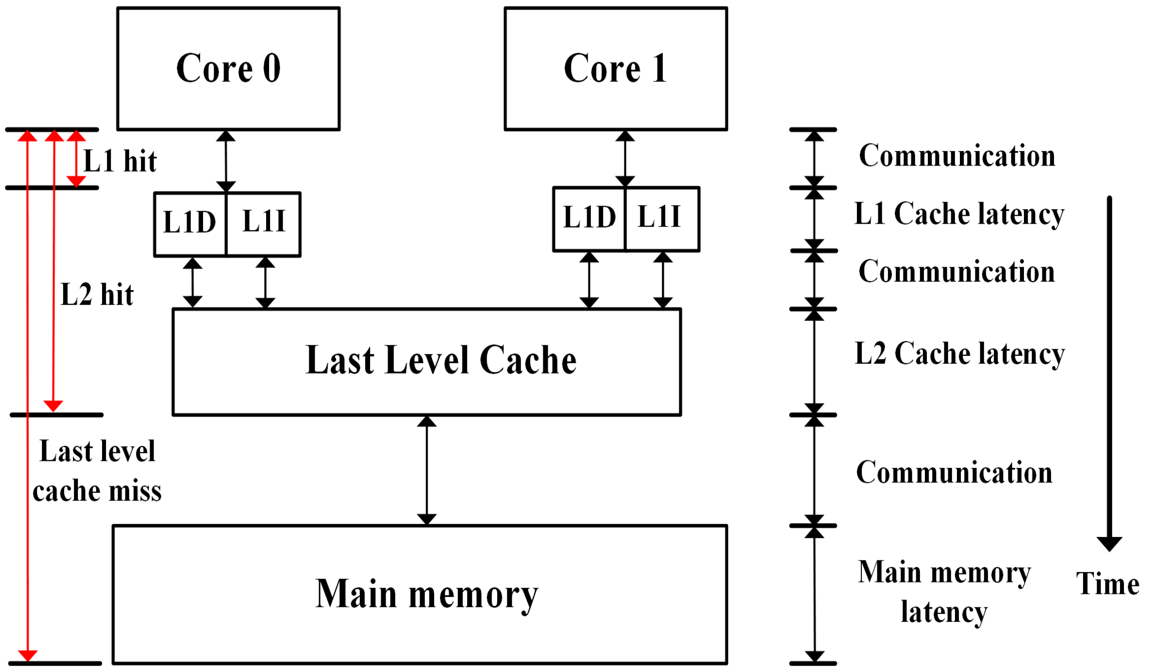


Figure 2.1: Sketch of CMP memory access that hits in the L1 cache, hit in the shared last level L2 cache and miss in both private L1 and shared L2 Cache

2.2.2 Cache Proposals for Multicores

In this section, we present few existing cache design proposals for CMP that are more pertinent to some part of our proposed work. These proposals focused on last-level cache in a CMP. The last-level of cache in a CMP can be private, shared or a hybrid. A private cache can be quicker compared to a shared cache and its content is not modified by other processors, while a shared cache can use the cache space in a superior manner as an application that functions on a huge data set will employ a bigger amount of the cache space when run concurrently with an application that is being executed on a small data set. A hybrid cache, on the other hand, merges the benefits of the private and shared caches. Several of these proposals are combinations of both private & shared cache organization. These hybrid proposals merge the benefits of both private and shared caches. Figure 2.2, puts forth three different manners of segregating a cache in CMPs with shared cache. In Figure 2.2 (a) each set belongs to only single core in CMP.

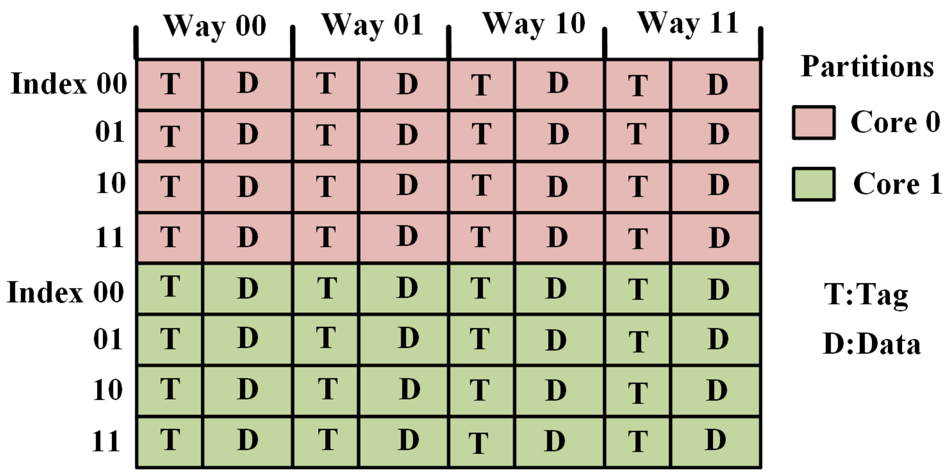


Figure 2.2 (a): Fixed sets per processors

While Figure 2.2(b) shows that each set contains equal number of cache blocks from two active cores in CMP. Whereas, the third case presents another mechanism in which, each set can be segregated with a variable number of cache blocks from two different cores as shown in Figure 2.2(c).

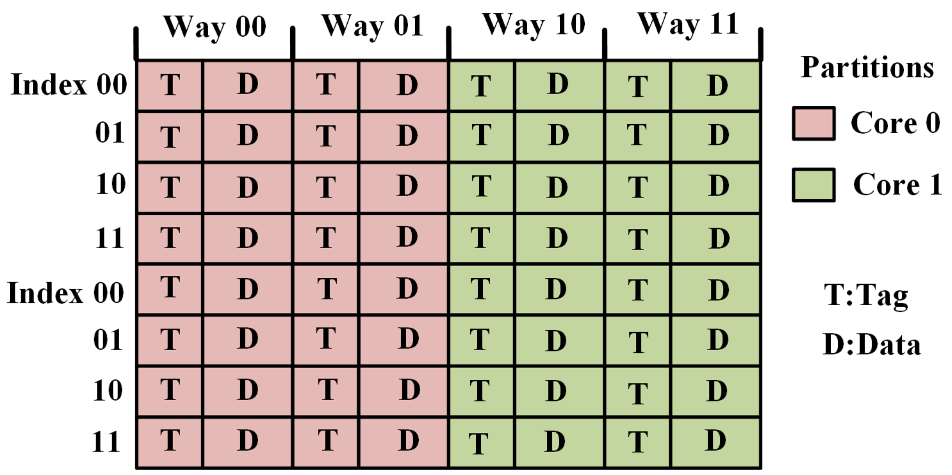


Figure 2.2 (b): Fixed Partitioned sets

It is easier to re-segregate the cache as there is no modification to which a group addresses maps to. Modifying the segregated size is intricate on account of the hashing function that maps memory addresses to sets has to be altered and the cache blocks have to be relocated or invalidated. Researchers have proposed several schemes that partitions the cache in a better manner so that, the core that can use additional cache space get more space whereas core that is not utilizing the pre-fetched blocks in the cache can get less cache space.

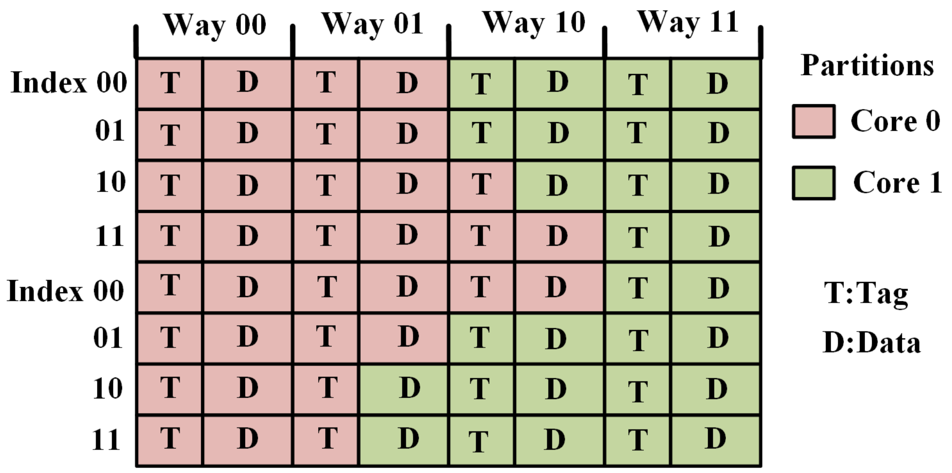


Figure 2.2 (c): Sets with variable number of blocks

The number of cores being integrated on the die is on the rise as CMP platforms are becoming popular. To lower the off-chip memory access, the last level cache is generally arranged to be a distributed shared cache. So as to evade hot-spots, cache lines are interleaved across the distributed shared cache slices. On the other hand, as one increases the number of cores and cache slices in the platform, majority of the data references are transmitted to the remote cache slices, thus increases the access latency to a considerable level. A hybrid last level cache was recommended by Zhao et al. [53]. On each cache slice, it has some degree of private space and some degree of shared space. The aim is to offer more hits into the local slice while trying to sustain a lower general miss rate for workloads with no sharing. The aim on the other hand, for workloads with adequate sharing is to permit additional sharing in the last-level cache slice. The researchers also discussed the hybrid last-level cache design choices and analyzed its hit/miss rate for several crucial server applications and multi-programmed workloads. As per the simulation outcomes it was inferred that this kind of architecture was most beneficial as it could increase the local hit rate to a great extent and simultaneously ensure that the overall miss rate was comparable to that of the shared cache. This scheme overlooks the matter related to the proportions of private as against the shared cache dynamic partitioning based on the workload behavior.

The issue of segregating a shared cache amongst several simultaneous running applications was analyzed by Qureshi et al. [59]. The frequently employed LRU strategy totally segregates a shared cache as and when demanded, providing more cache resources to the application with a higher demand and lesser cache resources to applications with lower demands. On the other hand, a higher need for cache resources is not always linked to a superior performance by the extra cache resources. It is advantageous for performance to use the cache resources in the application that can best make use of the resources instead of application that demands additional cache resources. Thus, the author recommend utility-based cache partitioning (UCP), a low-overhead, runtime method that partitions a shared cache between several application based on the lowering in cache misses that all the applications probably get for a given extent of cache resources. The recommended method observes each application at runtime employing a novel, cost-efficient, hardware circuit that needs storage less than even 2KB. The data gathered by the monitoring circuits is employed by a partitioning algorithm to choose the amount of cache size to be apportioned for each application. The assessment with 20 multi-programmed workloads indicates that UCP enhances functioning of a dual-core system by around 23% and on average 11% in contrast to LRU-dependent cache partitioning. The current study has overlooked the multi-threaded workload and the difference in utility of private data of rival threads.

A simple architectural extension and adaptive strategies for handling the L2 and L3 cache hierarchy in a CMP system was proposed by Speight et al. [55]. Specifically, the researchers assess two methodologies that enhance cache efficacy. Initially they recommended the employment of a small history table to offer clues to the L2 caches as to which lines are resident in the L3 cache. They use this table to remove few unrequired clean write backs to the L3 cache, lowering pressure on the L3 cache and on the on-chip bus. Next, they analyze the functioning advantages of permitting write backs from L2 caches to be transmitted to the adjacent on-chip L2 caches instead of compelling them to be grasped by the L3 cache. In addition to lowering the capacity stress on the L3 cache it also makes the following access quicker as L2-to-L2 cache transfers characteristically have lower latencies compared to accesses to a huge L3 cache array. The performance enhancement of these two schemes, and their merged impact, on four commercial workloads is the lowering in the overall execution time of around 13%.

Hardware-managed coherent caches and software-managed streaming memory are the two primary models for the on-chip memory in CMP systems. A direct comparison of the two models has been undertaken by Leverich et al. [56] assuming a similar group of presumptions pertaining to technology, area, and computational skills.

The aim is to enumerate how and when they vary in context of execution, consumption of energy, and width requirements in addition to latency tolerance for a CMPs. They show that for all data-parallel applications, the performance and scaling of the both cache-based and streaming models are similar. For specific applications that have limited data reuse, streaming scales are superior on account of superior bandwidth employment and macroscopic software pre-fetching. On the other hand, the initiations of methods like hardware pre-fetching and non-allocating stores to the cache-based model reduce the streaming benefit. Overall, the outcomes show that there is no adequate benefit in developing streaming memory systems where all on-chip memory structures are handled explicitly. However, the author indicates that streaming at the programming model level is especially advantageous, even with the cache based model, as it improves locality and develops chances for maximization of bandwidth. Furthermore, the author researcher notices that stream programming is really effortless with the cache-based prototype as the hardware ensures suitable, best-endeavor implementation even when the programmer fails to normalize the code of the application.

The Cooperative Cache Partitioning (CCP) to assign cache resources between threads running simultaneously on CMPs was put forth by Chang et al. [57]. Distinct cache partitioning schemes that employ a sole spatial partition recurrently all through a stable program stage, CCP resolves cache contention with several time-sharing partitions. Timesharing cache resources between partitions permits each thrashing thread to quicken noticeably in at least one segment by one-sidedly reducing the capacity assignments to other threads and also enhancing fairness by providing varied partitions an equal chance for execution. Time-sharing based cache partitioning is additionally merged with CMP cooperative caching [58] to develop the advantages of LRU-based latency optimizations, which result to a basic partitioning algorithm and superior execution for workloads that fail to take advantage of the cache partitioning. The author assess the efficacy of CCP by simulating a 4-core CMP running all grouping of 7 representative SPEC2000 benchmarks. For workloads that can take advantage of cache partitioning, CCP attains around 60%, and on average 12%, superior performance compared to the comprehensive seeking of optimal static partitions. Generally, CCP offers the most superior outcomes on almost all assessment criteria for varied cache sizes.

The huge data working sets of commercial and scientific workloads underline the L2 caches of CMPs. Few CMPs employ a shared L2 cache to increase the on-chip cache storage and reduce off-chip misses. Other CMPs employ private L2 caches and duplicates data to restrict the delay on account of global wires and reduce cache access time. The latest hybrid schemes employ selective duplication to balance latency and capacity, but their static duplication norms may lead to performance degradation for some amalgamations of workloads and system configurations. The Adaptive Selective Replication (ASR) has been recommended by Beckmann et al. [60]; it is a method that dynamically oversees workload behavior to control duplication. ASR duplicates cache blocks only when it evaluates the advantage of duplication (lower L2 hit latency) to surpass the outlays (additional L2 misses). Full-system simulations of 8-processor CMPs indicate that ASR offers a healthy execution: enhancing the execution by over 29% in contrast to shared caches, 19% in contrast to private caches and 12% in contrast to CMP-NuRapid and Victim Replication.

A comprehensive research of fairness in cache sharing amongst threads in a chip multiprocessor (CMP) architecture was put forth by Kim et al. [61]. The earlier studies related to CMP architectures have merely analyzed throughput maximization methods for a shared cache. Researchers have not assessed the problem of fairness in cache sharing, and its association to throughput. Fairness is an essential problem as the Operating System (OS) thread scheduler's efficacy relies on the hardware to offer a suitable fair cache sharing to co-scheduled threads. In the absence of such hardware, grave issues, including thread starvation and priority inversion, may occur making the OS scheduler unproductive. The researcher provides many inputs. Initially, the researcher recommends and assesses five cache fairness metrics that gauge the extent of fairness in cache sharing, and indicates that two of them are linked strongly with the execution -time fairness. Execution time fairness is described as how uniform the execution times of co-scheduled threads are modified; where each modification is comparative to the execution time of the same thread being implemented solely. Next, using the metrics, the researcher recommends static and dynamic L2 cache partitioning algorithms that maximize fairness. It is effortless to implement the dynamic portioning algorithm as it does not need any major profiling and has a reduced overhead; it does not limit the cache replacement algorithm to LRU. Despite the static algorithm require the cache to keep LRU stack information, it can help the OS thread scheduler to evade cache thrashing. Finally, the author studies the relationship between fairness and throughput, while maximizing throughput does not necessarily improve fairness. Employing a group of co-scheduled pairs of applications (benchmarks), on average the recommended algorithms enhance fairness by factor of 4x while enhancing the throughput by 15%, in contrast to a non-partitioned shared cache.

A distributed L2 cache management approach via page-level data to cache slice mapping in a processor chip comprising of several cores was recommended and analyzed by Jin et al. [62]. L2 cache handling is an essential multi-core processor design facet to overpower non-uniform cache access latency to achieve high performance during the execution of the program and to lower on-chip net-work traffic and its power consumption.

An arrangement for the on-chip memory system of a chip multiprocessor, in which a 16MB pool of 256 L2 cache banks is shared by 16 processors, was recommended by Huh et al. and Foglia [63, 65]. The L2 cache is arranged as non-uniform cache architecture (NUCA) array with a switched network inserted in it for superior performance. Researchers indicate that this arrangement can endorse the range of degrees of sharing: unshared, in which every processor owns a private segment of the cache, thereby, lowering the hit latency; completely shared, in which each processor shares the entire cache, thereby reducing misses, and every point in between. Researchers seek the best level of sharing for several cache bank mapping strategies, and also assess a per-application cache partitioning policy. They infer that a static NUCA arrangement with sharing degrees of two or four is most suitable for varied commercial and scientific parallel workloads.

A dynamic cache partitioning scheme that clearly assigns cache space between concurrently executing process and reduces the overall cache misses was put forth by Suh et al. [66]. Employing a group of on-line counters, the scheme dynamically estimates each process gain or loss in varied cache assignments in context of the number of cache misses. Then, the dynamic alteration of the allocation occurs to ensure that more essential processes can employ additional cache space [67].

Nahalal, a new CMP cache architecture that partitions the L2 cache as per the data sharing of the programs was recommended by Guz et al. [68]; this provides locality of reference to shared as well as private data. A part of the L2 memory is located in the center of the chip, surrounded on all sides by all processors, while the remainder of the L2 memory is situated on the outer slices [20]. The hottest shared data populates the inner memory and is quickly accessed by all the processors. A "backyard" for each processor is created by the outer slices.

The cache-fair scheduling algorithm, a novel operating system scheduling algorithm for multi-core processors is introduced by Fedorova et al. [69]. This algorithm lowers the impact of unequal CPU cache sharing that take place on these processors and result in partial CPU sharing, priority inversion, and insufficient CPU accounting. As per the author, the execution of the algorithm in the Solaris operating system indicates that it generates better priority enforcement and enhanced execution stability for applications. With traditional scheduling algorithms, application performance on multicore processors differs by around 36% based on the runtime attributes of concurrent processes. The author assessed the execution of the algorithm in Solaris 10 and indicated that it crucially lowers co-runner dependent performance difference, while levying slight drawback on best-effort threads. Co-runner-dependent performance is the outcome of unequal cache sharing, and by evading the same, the researchers deal with the issues that were the result of unequal cache sharing.

Uncontrolled sharing in CMP results in situations where one core eliminates beneficial L2 cache content belonging to another core. To deal with this issue, Tam et al. [70] executed a software tool that permitted partitioning of the shared L2 cache by directing the assignment of physical pages. This method is successful in lowering cache contention in multi-programmed SPECcpu2000 and SPECjbb2000 workloads. Performance enhancements of around 17% were attained without any negative impact on co-scheduled applications. This study failed to analyze how this method dynamically altered the number of partitions accorded to an application in an on line, that too in a reduced overhead conduct.

As many schemes already exists and there is a need to find an efficient dynamic partitioning scheme that explicitly allocates cache space amongst simultaneously executing tasks this research work proposes to investigate the cache allocation that can be dynamically changed so that more needy tasks can use more cache space and also propose to investigate methods to resolve ownership of cache space efficiently.

As a response to the rising (comparative) wire delay, different methods have been proposed by architects to handle the influence of slow wires on huge uniprocessor L2 caches. Block migration (e.g., D-NUCA and NuRapid) lowers the average hit latency by transferring commonly employed blocks towards the lower-latency banks. Transmission Line Caches (TLC) employs on-chip transmission lines to offer low latency to all banks. Conventional stride-based hardware prefetching attempts to endure instead of lowering the latency. There are more issues with chip multiprocessors (CMPs). To begin with, CMPs frequently share the on-chip L2 cache, needing several ports to offer adequate bandwidth. Next, multiple threads indicate several varied working sets, which vie for restricted on-chip storage. Thirdly, sharing code and data interferes with block migration, as one processor's low-latency bank acts as another processor's high-latency bank. L2 cache designs for CMPs that merge these three latency management methods were proposed by Bradford et al. and Kannan [71]. The researchers employ comprehensive full-system simulation to evaluate the performance tradeoffs for both commercial and scientific workloads.

The probability of using a very small data cache, split for fulfilling the needs of the temporal and spatial streams was analyzed by Naz et al. [72].

The influence of different cache architectures on the execution behavior of multi-threading applications was analyzed by Tao et al. [74]. His emphasis was on four common cache planning problems: cache structure, configuration criteria, coherence influence, and prefetching strategies. The research relies on a self-developed cache simulator that designs the operability of a multi-core cache hierarchy with arbitrary levels and different organizations. Both the hardware and program developers can be directed by the attained outcomes to maximize their cache designs or the program codes.

In a shared L2 cache model of CMP, cache coherency is an important research issue to be addressed. Although traditional cache coherency protocol has been used [83]. In a CMP, cache coherency has been handled in a way to take advantage of its design structure. Roy et al. [75] proposes variable forwarding cache coherence to improve performance of the system by using variable forwarding. This work proposes and investigates various cache coherency issues that exist in CMP and various ways of resolving them.

2.3 An Introduction to Multiprocessor Memory Consistency

Serial programs running on Von-Neumann machines present a simple intuitive model to the programmer. The instructions seem to be executed in the manner stated by the programmer or compiler irrespective of the fact that the design of the machine really executes them in a varied sequence. Crucially, a program's load returns the last value that was written in the memory location. Similarly the value of the next load is ascertained by the store to a memory location. This description results in a direct implementation and semantics for programs being executed on a single uniprocessor. Multi-threaded programs being executed on multiprocessor machines obscure the programming model and also the implementation to enforce a specific model. Specifically, the value returned by a given load is indistinct as the latest store may have taken place on a varied processor core. Hence, architects describe memory consistency models [76] to state how a processor core can detect memory accesses from other processor cores in the system. Sequential consistency is a model described to be one that the outcome of any execution is similar as if the operations of all processors were executed in some chronological order, and the operations of each distinct processor act in this sequence in the order stated by its program [77]. Other, more relaxed consistency models [78] can provide the system designer additional freedom to further optimize memory system to decrease memory latency. For instance, a relaxed model (memory) allows simple implementation of write buffers with the bypassing option. While relaxed prototypes can enhance performance by retiring memory instructions prior to them having being noticed by other processors in the system, proper synchronization of multi-threaded applications is still needed. Systems employing relaxed memory consistency prototype either have additional instructions that permit a programmer to compel orderings between loads and stores [79], or describe semantics in a way that a programmer can synchronize by employing sensibly developed series of loads and stores. The addition of cache memories influences how consistency is enforced irrespective of sequential or relaxed consistency.

2.3.1 Effect of Caches on Memory Consistency

Cache memories have been paramount in facilitating the rapid performance progress of microprocessors over the past twenty years. They allow processor speeds to increase at a greater rate than DRAM speeds by exploiting locality in memory accesses. The importance of caches is their effective operation with very little impact on the programmer or compiler. In other words, details of the cache hierarchy do not affect the instruction set architecture and their operation is all hardware-based and automatic from a programmer’s point-of-view. While implementing a cache hierarchy had little ramification on a uniprocessor’s memory consistency, caches complicate multiprocessor memory consistency. The root of the problem lies in store propagation. Figure 2.3 illustrates a simple example of incoherence. Initially, memory location A has the value 42 in memory, and then both Core 1 and Core 2 loads this value from memory into their respective caches. At time 3, Core 1 increments the value at memory location A from 42 to 43 in its cache, making Core 2’s value of A in its cache stale or incoherent. To prevent incoherence, the system must implement a cache coherence protocol to regulate the actions of the cores such that Core 2 cannot observe the old value of 42 at the same time that Core 1 observes the value 43. The design and implementation of these cache coherence protocols are the main topics of discussion. While R1 and R2 – the two cores in a system may load the same memory block into their corresponding private caches, a following store by any one of the cores would result in a variation in the values of the caches. Hence, if R1 stores to a memory block that exists in both the caches of R1 and R2, R2's cache has a probable old value on account of the R1's default function of storing its individual cache. The current cache incoherence would not be an issue if R2 never loads to the block while still cached or if the multiprocessor did not back the transparent shared-memory abstraction. However, since the multiprocessor memory model should support shared-memory programming, at some point the future loads of the block by R2 needs to obtain the novel value stored by R1, as described by the model. Thus, R1's store must probably impact the status of the cache line in R2's cache to sustain consistency, and the methods for doing the same are known to be cache coherence. Hence, the policy of the current study considers the cache coherence to be an independent problem related to memory consistency that is essential but not adequate to implement a given model. All the protocols that we discuss can endorse any memory consistency prototypes, but our explanations would presume sequential consistency.

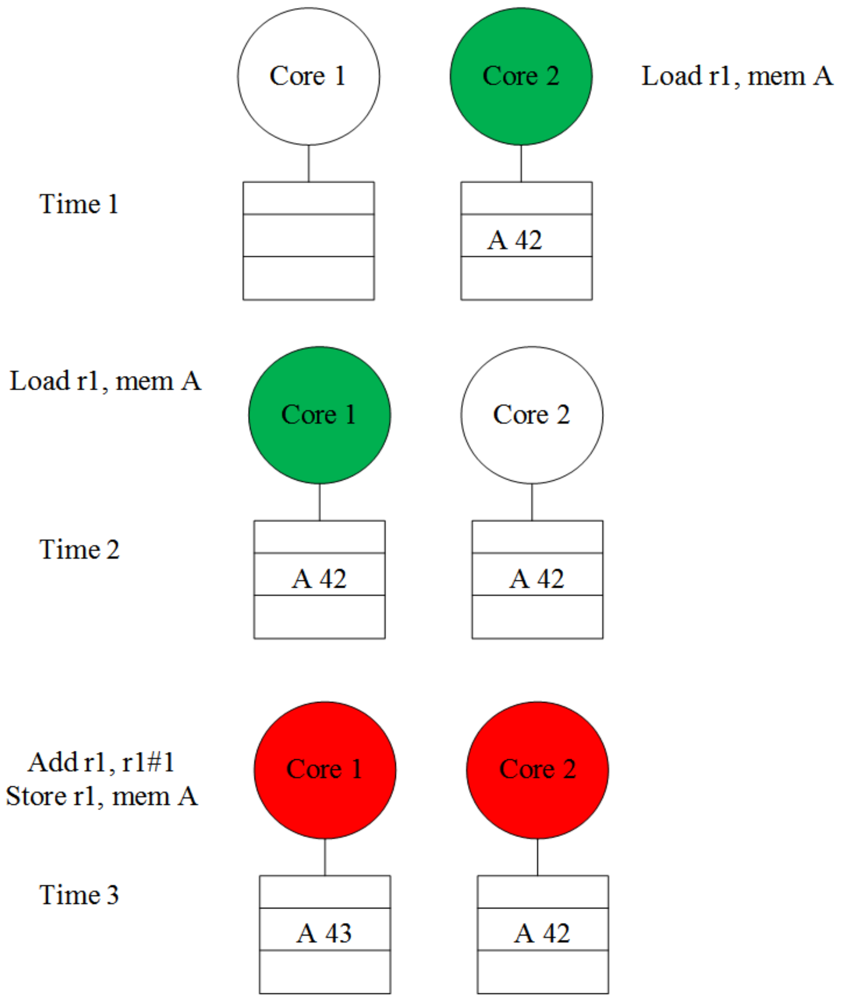


Figure 2.3: Problem of Incoherence

2.3.2 Cache Coherence Invariant and Permission

The example of an incoherent situation described in the previous section 2.2 is intuitively “incorrect” in that cores observe different values of a given datum at the same time. In this section, we transition from an intuitive sense of what is incoherent to a precise definition of coherence. There are several definitions of coherence that have appeared in textbooks and in published papers. We present the definition we prefer for its insight into the design of coherence protocols. In the sidebar, we discuss alternative definitions and how they relate to our preferred definition. The basis of our preferred definition of coherence is the single-writer–multiple-reader (SWMR) invariant. There may be either a single core that may write (and may also read) or multiple cores that may read any given memory location at any given moment of time. Thus, there is never a time when a given memory location may be written by one core and simultaneously either read or written by any other cores. Another way to view this definition is to consider, for each memory location, that the memory location’s lifetime is divided up into epochs. In each epoch, either a single core has read–write access or some number of cores (possibly zero) have read-only access. Figure 2.4 illustrates the lifetime of an example memory location, divided into four epochs that maintain the SWMR invariant.

In addition to the SWMR invariant, coherence requires that the value of a given memory location is propagated correctly. To explain why values matter, let us reconsider the example in Figure 2.2. Even though the SWMR invariant holds, if during the first read-only epoch Cores 2 and 5 can read different values, then the system is not coherent.

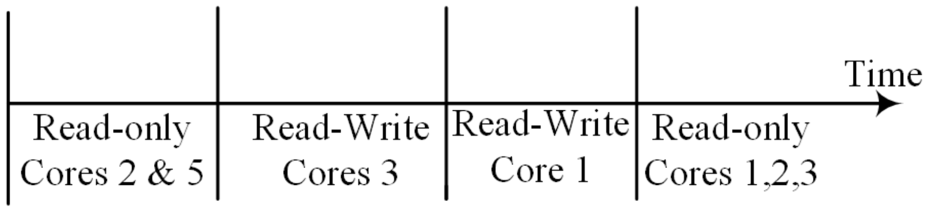


Figure 2.4: Dividing a given memory location’s lifetime into epochs

Similarly, the system is incoherent if Core 1 fails to read the last value written by Core 3 during its read–write epoch or any of Cores 1, 2, or 3 fail to read the last write performed by Core 1 during its read–write epoch. Thus, the definition of coherence must augment the SWMR invariant with a data value invariant that pertains to how values are propagated from one epoch to the next. This invariant states that the value of a memory location at the beginning of a period is similar to the value of the memory location at the completion of its last read–write period. There are other interpretations of these invariants that are equivalent. One notable example [88] interpreted the SMWR invariants in terms of tokens. The invariants are as follows. For each memory location, there exist a fixed number of tokens that is at least as large as the number of cores. If a core has all of the tokens, it may write the memory location. If a core has one or more tokens, it may read the memory location. At any given time, it is thus impossible for one core to be writing the memory location while any other core is reading or writing it.

2.3.3 Coherence invariants

1. Single-Writer, Multiple-Read (SWMR) Invariant. At any given (rational) time, for any memory location B, only a single core is present that may write to B (and also has the ability to read it) or some limited cores that may only read B.

2. Data-Value Invariant. The value of the memory location at the initiation of a period is equivalent to the value of the memory location at the completion of its last read–write period.

2.4 Coherence Protocols

The goal of a coherence protocol is to maintain coherence by enforcing the invariants introduced in the previous section. To implement these invariants, we associate with each storage structure (each cache) and the LLC/memory a finite state machine called a coherence controller.

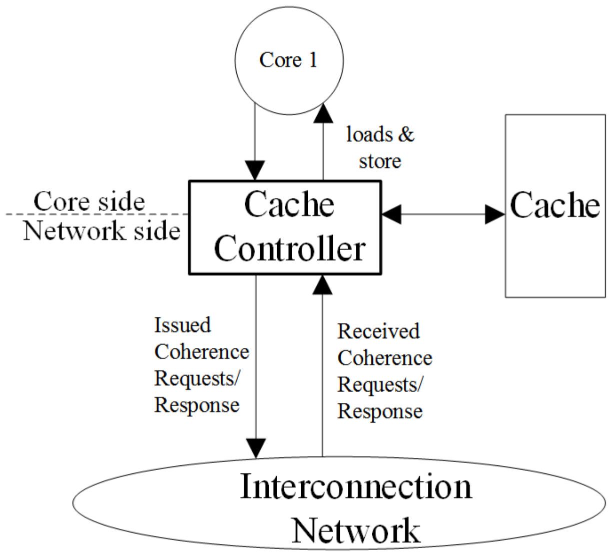


Figure 2.5: Cache Controller

The collection of these coherence controllers constitutes a distributed system in which the controllers exchange messages with each other to ensure that, for each block, the SWMR and data value invariants are maintained at all times. The interactions between these finite state machines are specified by the coherence protocol. Coherence controllers have several responsibilities. The coherence controller at a cache, which we refer to as a cache controller, is illustrated in Figure 2.5. The cache controller must service requests from two sources. On the "core side," the cache controller interfaces to the processor core. The controller accepts loads and stores from the core and returns load values to the core. A cache miss causes the controller to initiate a coherence transaction by issuing a coherence request (e.g., request for read-only permission) for the block containing the location accessed by the core. This coherence request is sent across the interconnection network to one or more coherence controllers. A transaction includes a request and the other message(s) that are exchanged in order to satisfy the request (e.g., a data response message sent from another coherence controller to the requestor). The types of transactions and the messages that are sent as part of each transaction depend on the specific coherence protocol. On the cache controller's "network side," the cache controller interfaces to the rest of the system via the interconnection network. The controller receives coherence requests and coherence responses that it must process. As with the core side, the processing of incoming coherence messages depends on the specific coherence protocol.

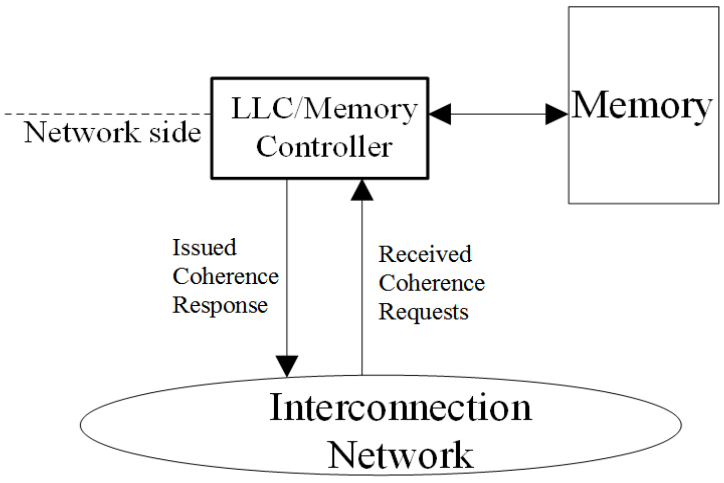


Figure 2.6: Memory Controller

The coherence controller at the LLC/memory, which we refer to as a memory controller, is illustrated in Figure 2.6. A memory controller is akin to a cache controller, the sole exception being that it generally has only a network side. As such, it does not issue coherence requests (on behalf of loads or stores) or receive coherence responses. Other agents, such as I/O devices, may behave like cache controllers, memory controllers, or both depending upon their specific requirements. Each coherence controller implements a set of finite state machines rationally one independent, but similar finite state machine per block and receives and processes events (e.g., incoming coherence messages) depending upon the block’s state. For an event of type E (e.g., a store request from the core to the cache controller) to block B, the coherence controller takes actions (e.g., issues a coherence request for read-write permission) that are a function of E and of B’s state (e.g., read only). After taking these actions, the controller may change the state of B. As stated initially in the current study, permitting multiple cores to access the same address space to store data in their private caches leads to a cache coherence problem. This problem is made transparent to software via hardware implementation of cache coherence protocols. There are two varied strategies that can be employed to sustain cache coherence, and depending on them, we can segregate amongst invalidation and update-based cache coherence protocols [80, 81]. On getting a write request, invalidation-based protocols [78] sends invalidation messages to all the sharers (the sole exception being the requester) and it requires privates copies must be invalidated. However, update based protocols forwards the newly written copy to all the sharers after write operation. The chief drawback of the update-based protocols is the generation of heavy network traffic. This is more evident, when a processing core writes a block several time prior to another core reading the block; this results in all updates being notified, requiring a varied message for each one. Despite this disadvantage being lowered by adaptive protocols [82], this is one of the chief reason why the latest systems employ invalidation-based protocols and, thus, the current work considers invalidation based cache coherence protocols. Invalidation-based protocols need to guarantee the subsequent invariant.

Logically, at any point, a single core can write a cache block or multiple cores (SWMR) can read one cache block. Thus, if a processing core desires to alter a cache block, this block has to be invalidated beforehand (the read permission needs to be invalidated) from the other caches. Similarly, if a processing core desires to read a cache block, the write permission allotted to some other cache needs to be invalidated beforehand. There are other crucial design choices that impact the ultimate efficacy of the protocol when implementing a cache coherence protocol. The subsequent section outlines the description of a cache coherence protocol and subsequently discusses the existing protocols.

2.4.1 Design space for cache coherence protocols

There are many options for designing cache coherence protocols based on the states of the blocks present in the private caches. These options have been generally termed based on the states they use: MOESI, MOSI, MESI, MSI etc. Each state stands for varied authorizations for a block present in a private cache.

2.4.2. Specifying cache coherence protocols

A designer of a coherence protocol must choose the states, stable states, transient states, transactions, events, and transitions for each type of coherence controller in the system.

Cache block states:

In a system with only one actor (e.g., a single core processor without coherent DMA), the state of a cache block is either valid or invalid. There might be two possible valid states for a cache block if there is a need to distinguish blocks that are dirty. A dirty block has a value that has been written more recently than other copies of this block. For example, in a two-level cache hierarchy with a write-back L1 cache, the block in the L1 may be dirty with respect to the stale copy in the L2 cache. A system with multiple actors can also use just these two or three states, but we often want to distinguish between different kinds of valid states. There are four characteristics of a cache block that we wish to encode in its state: validity, dirtiness, exclusivity, and ownership [83].

The latter two characteristics are unique to systems with multiple actors.

Validity: A valid block has the most up-to-date value for this block. The block may be read, but it may only be written if it is also exclusive.

Dirtiness: As in a single core processor, a cache block is dirty if its value is the most up-to-date value, this value differs from the value in the LLC/memory, and the cache controller is responsible for eventually updating the LLC/memory with this new value. The term clean is often used as the opposite of dirty.

Exclusivity: A cache block is exclusive if it is the only privately cached copy of that block in the system (i.e., the block is not cached anywhere else except perhaps in the shared LLC).

Ownership: A cache controller (or memory controller) is the owner of a block if it is liable for replying to coherence requests for that block. In most protocols, there is exactly one owner of a given block at all times. A block that is owned may not be removed from a cache to permit another block to enter due to a capacity or conflict miss without giving the ownership of the block to another coherence controller. Non-owned blocks may be evicted silently (i.e., without sending any messages) in some protocols.

In this section, we first discuss some commonly used stable states of the blocks that are not currently in the midst of a coherence transaction and then discuss the use of transient states for describing blocks that are currently in the midst of transactions.

2.4.3 Stable States

Many coherence protocols use a subset of the classic five state MOESI model first introduced by Sweazey and Smith [83]. These MOESI (often pronounced either “MO-sey” or “mo-EE-see”) states refer to the states of blocks in a cache, and the most fundamental three states are MSI; the O and E states may be used, but they are not as basic. Each of these states has a different combination of the characteristics described above.

M (modified): In CMP system, only a private cache of a single core keeps the valid copy of the data block in this state, and only this single core has permission to read/write over the block. The private caches of the other cores do not hold any valid copy of this block. Even the shared L2 cache holds an invalid copy of this block. In the case of requests from other cores for this particular block, the private cache with valid copy of the block in modified state must provide requested block

S (shared): In this state cache holds a valid data block. In CMP system, multiple cores are allowed to keep private copies of the data block in shared state but a single core holds the block in owned state. If there is no private cache with data block in owned state then shared L2 cache is responsible for providing the requested block.

I (invalid): In this state cache do not keeps a valid copy of the requested data block. A valid copy of the requested data block can be present in shared L2 cache or in the private cache of another core.

O (owned): In this state, the copy of the block in the cache is valid as well as dirty but it is not the exclusive copy. The private caches of the other cores may hold a read-only copy of this block but none of them can hold the block in owned state. The shared L2 cache holds a stale copy of the block. In case other cores need to modify this block, the coherence controller needs to send invalidation messages to invalidate all the private copies in the system. This scenario is quite similar with the shared state. The main difference between these two states is that in case of a miss, the private cache with block in owned state is responsible for forwarding this block since the shared L2 cache holds the stale copy of the block. However, the block evictions in the owned state entails write back operations.

E (exclusive): In this state cache holds a valid copy of the requested cache block. This single core is allowed to read/write to this valid copy of the data block. Another valid copy of the block may exist in the shared on-chip cache.

The most basic protocols use only the MSI states, but there are reasons to add the O and E states to optimize certain situations. When we present MESI snooping and directory protocols in later chapters, we discuss the issues involved.

2.4.4 Transient States

Thus far we have discussed only the stable states that occur when there is no current coherence activity for the block, and it is only these stable states that are used when referring to a protocol (e.g., “a system with a MESI protocol”). However, there may exist transient states that occur during the transition from one stable state to another stable state. We had the transient state IVD (in I, going to V, waiting for DataResp). In more sophisticated protocols, we are likely to encounter dozens of transient states. We encode these states using a notation XYZ, which denotes that the block is transitioning from stable state X to stable state Y, and the transition will not complete until an event of type Z occurs. For example, in a protocol in a later chapter, we use IMD to denote that a block was previously in I and will become M once a D(ata) message arrives for that block. The various combinations using a subset of these states are illustrated below to design different protocols:

MSI

The simplest cache coherence protocol requires at least MSI states (three states) to enforce invariant discussed previously while using write back private cache. A single core in the CMP has provided read/write permissions for a block, which means its private cache holds block in M (modified) state. However, other cores in the CMP have permissions to read the block while caching the block in S (shared state). Figure 2.7 presents state transition diagram for a simple MSI based cache coherence protocol.

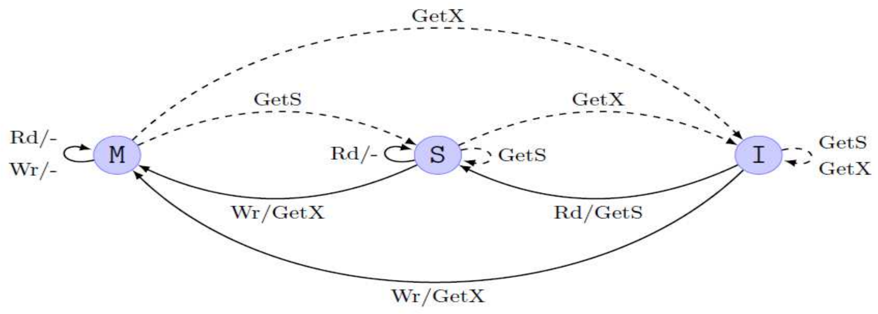


Figure 2.7: State Diagram for MSI

In case of a request for a new block from one of the processing core, an existing block must be evicted from the private cache to make space for the incoming block. Moreover, the eviction of block requires state transition to the I state but we have not shown these transitions in the state diagram.

In order to read a block the core must issues GetS request for the block while executing an application. It requires read permission (Rd/GetS) for the block if the block is not previously accessed. However, if the core already have a read permission then a read request is generated (Rd/-). On the other side, if the executing core needs to write to a cache block then it must generate GetX request to obtain the write permission (Wr). As shown in Figure 2.7, the transition due to the requests generated by remote cores are represented by dashed arrows while the normal arrows represents the transition caused by local requests. The design of MSI cache coherence protocols is relatively simple but it has few drawbacks which can be improved by adding few more states such as Exclusive (E) and Owner (O). The addition of exclusive (E) state further optimizes the simple MSI protocol for non-**shared data blocks.** Hence, it is essential to obtain good performance for sequential applications running on a multiprocessor. On receiving a read request from a core, the data block is brought into private cache and stored with exclusive state instead of the shared state. In this case, the requesting core obtains write permissions for a block. However, the subsequent write request for this block will not result into cache miss (in case none of the other cores requested for the block). After write operation the status of the cache block will be silently changed to the modified one. The difference between the exclusive state and modified state is that the data block with exclusive state is clean and the shared LLC cache holds a valid copy of the data block. The main benefit of this state is that in case of block eviction or read requests from other processing cores, there is no need to write back data to the shared cache.

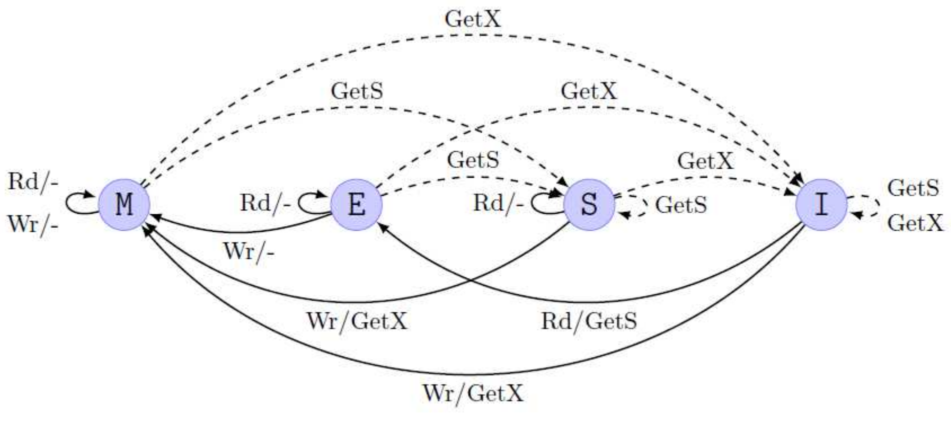


Figure 2.8: State transition for MESI

The state transition diagram for the MESI based cache coherence protocol is presented in Figure 2.8.

Now, to further optimize the MESI protocol [83] an additional state is introduced know as owned state. The main advantage of owned state is that in case of a shared request from a remote processing core to the cache block stored in the modified state, the state changes from modified to the owned state instead of transition to the shared state. The owned state is quite similar to the shared state with the difference that the shared LLC cache do not hold a valid copy of the data block. Following are the benefits of the addition of owned state:

• The first one is the reduction of network traffic because the processing core does not need to write the data block back to the shared L2 cache during a remote read request and the block state transitions from the modified state to the owned state.

• Secondly, the shared L2 cache is not required to maintain a replica for blocks within the owned state, which can lead to improved utilization and thus lowering the miss rate of the shared cache. Note that in CMPs with a shared cache organization, the misses of the shared cache require off-chip accesses.

• Thirdly, for a few architectures cache misses can be resolved more rapidly by supplying data from private caches as compared to the shared cache. This is primarily the case of the cc-NUMA machines [93], in which the shared cache is represented by main memory, or perhaps even CMPs utilizing a private cache organization [114]. However, in CMPs along with a shared cache organization, the data block can be delivered more quickly from the shared cache and, in this case, this benefit can be ignored. Figure 2.9 presents the state transition diagram for the MOESI cache coherence protocol.

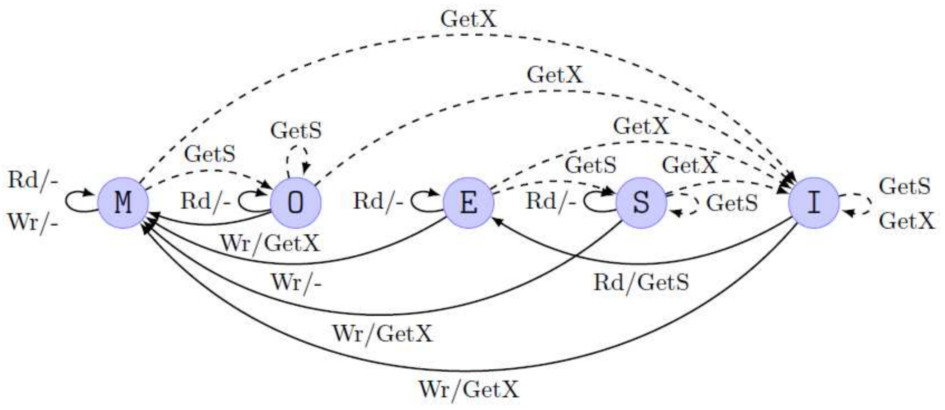


Figure 2.9: State Transition for MOESI Protocol

The owned state also can also be witnessed without the exclusive state resulting in a MOSI protocol whose state transition diagram is not indicated. All the cache coherence protocols evaluated in the current study presume MOESI states.

2.5 Existing Cache coherence protocols

In this section, we will present few already existing cache coherence protocols for multiprocessors.

2.5.1 Snooping bus-based coherence protocol

Goodman et al. [79] first described snooping coherence on bus. In snoop-based protocols, a coherence request is broadcast to all nodes and every node snoops the request. Each node maintains an identical state machine to implement the cache coherence protocol. By snooping the request, each node applies the message on current state of the state machine and responds accordingly. In these systems, a node is considered a uniprocessor with its private cache hierarchy. Snoop-based protocol is implemented by using different techniques depending on the topology for interconnect. The most interesting ones are bus-based and ring-based snooping. Bus-based snooping is the widely used approach for cache coherence where a bus connects all components to a set of wires. A bus offers the key ordering and atomicity attributes that allow straightforward coherence operations. Goodman [79] first described snooping coherence on a bus. This technique has some variants. The sent messages are viewed by all the endpoints on a bus in a similar total order. Busses offer atomicity such that at one time only one message is visible on the bus and that all endpoints see that message. Buses execute shared lines that permit any endpoint to alter a signal or condition that is visible across to all other endpoints during a bus transaction. Shared wires are used for bus arbitration. They are also used in coherence actions like a processor having a shared copy of the cache line can indicate whether there is any shared cache copy on snooping a GETS request in the bus. When a GETX request (permission to modify data) is introduced on the bus, all nodes snoop their caches and the memory controller gets ready to fetch the data from DRAM. If the tag is present in a processor’s cache in read-shared state S, the coherence state is altered to invalid state I to nullify read permission from its own processor. If the processor’s cache has a tag in altered state M, exclusive state E, or owner state O, it declares the shared owned line to constrain a memory reply and then puts the data on the bus before moving its state to I. In a bus-based protocol, the shared owned line provides the functionality of signaling the memory controller not to send the data when data is altered in a processor’s cache. If there is no processor to provide the data (shared owned line is not set) memory controller provides the data. When a GETS request (permission to read data) appears on the bus, all nodes snoop their caches and the memory controller gets ready to fetch the data from DRAM. If the processor's cache has a tag in M, E, or O state, it declares the shared owned line to constrain a memory reply and then puts the data on the bus before moving its state to O or S state depending on implementations; otherwise memory controller serves the data and requester moves to E- state.

Cache replacements are performed silently for copies in S or E state. For M or O state it requires a write back of modified data to memory. To write back, the node needs to introduce a WRITEBACK bus transaction that contains includes the data and is accepted by memory. The atomic character of the bus guarantees that racing coherence requests are ordered in context to the write back function. Sometimes the write back data may be buffered in write back buffer to serve the misses before writing back. In that case, bus snoops must also look into write back buffer. Bus arbitration determines the fairness of the bus-based broadcast protocol as a processor can complete its transaction if and only if it is able to send its request on the bus. A state transition must appear atomic. For example, two nodes may send the UPGRADE request simultaneously while both are in shared state. One of them will get the bus exclusive access and the other will not. The bus-owner node's UPGRADE will invalidate second requester's copy. Once the second requester gets the bus access, the UPGRADE message is no longer valid as it does not have a shared copy. There are several ways bus can be implemented. One approach is to use electrically shared wires which are held exclusively for the entire cache coherence transaction (atomic transaction). A better performing option is to use split transactions to permit other processors to get the bus while awaiting a reply. Modern snooping systems execute a logical bus employing additional switches, state, and logic instead of shared electrical wires; some of those systems also execute the ordering of a bus only for coherence control messages and not for data such as Sun Starfire [86] [87] system executes a logical bus merely for coherence request messages, but data responses are transmitted on a different switched interconnect. Few of the buses use pipelining methods in order to attain more concurrency. While these more aggressive buses may ease the atomicity attribute, they still offer a total order of coherence requests that allows a straightforward execution of snooping.

The disadvantage of these bus-based snooping protocols is that the buses have limited bandwidth. The more often it snoops on the bus, the less bandwidth is available for the bus's main job of transferring information back and forth. In addition, the broadcast nature of cache messages requires even more valuable bandwidth.

2.5.2 AMD-Hammer Coherence protocol

The Opteron systems from AMD made use of Hammer cache coherence protocol for CMPs [8] [9]. Just as snooping-based protocols, Hammer fails to retain any coherence details about the blocks kept in the private caches and, due to this fact, it relies upon broadcasting requests to each individual cores on the chip to resolve cache misses. Its key benefit in comparison with snooping-based protocols is that it manages systems which makes uses unordered point-to-point interconnection networks. The hammer protocol supports small to moderate number of cores and it works with unordered interconnection network where traditional snooping is not possible.

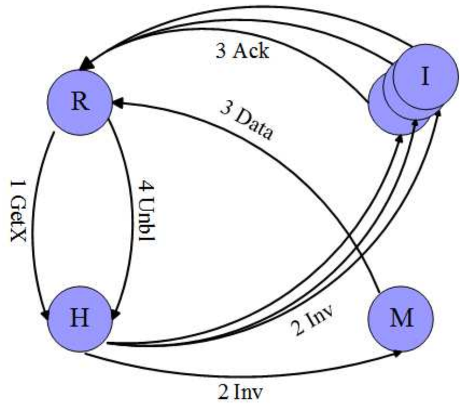


Figure 2.10: Cache to Cache miss in AMD-Hammer protocol

In case of a cache miss, the hammer first sends a request to the home memory, it allocates a transaction entry to place the block into a busy state and the request is send to all the cores within the system like broadcast based protocol to obtain the requested block and to clear away the potential copies of the block in case of a write miss. Finally, the request is forwarded to the memory controller that fetches data from main memory and sends it to the requester. After receiving forwarded request, each core sends an explicit acknowledgment or the data message to the requester. As soon as the requester obtains each of the responses, it transmits an unblock message to the home tile. This message informs the home tile with the fact that the miss has already been fulfilled. In such a manner, if there is an additional request for the identical block waiting with the home tile, then it is processed by providing the requested block. Despite the fact that the unblock message may also introduce additional contention towards the home tile, it really stops the appearance of race problems. This message is also helpful to eliminate race conditions in directory-based protocols, which are discussed shortly. Figure 2.10 demonstrates an illustration of how Hammer resolves a cache-to-cache transfer miss. As shown in figure, the core R communicates a GetX request (write) towards the home node (H). Thereafter, home node transmits invalidation messages to all the cores. The core having the ownership of the requested block replies with the data block (3 Data). On the other hand, all the cores that do not maintain a copy of a given block (Invalid) retort by means of the acknowledgement messages. As soon as the requester obtains each of the responses, it transmits the unblock message (4 Unbl) towards the home core. At first, we observed that, this protocol requires three hops within the critical path before the required data block is acquired. Secondly, transmitting the invalidation messages raises significantly the traffic inserted within the interconnection network and, as a consequence increases power consumption.

2.5.3 Token-Based Coherence protocol

Token coherence [88] is a framework for designing coherence protocols whose main asset is that it decouples the correctness substrate from several different performance policies. Token coherence protocols can avoid both the need of a totally ordered network and the introduction of additional indirection caused by the access to the home tile in the common case of cache-to-cache transfers. Token coherence protocols keep cache coherence by assigning T tokens to every memory block, where one of them is the owner token. Then, a processing core can read a block only if it holds at least one token for that block and has valid data. On the other hand, a processing core can write a block only if it holds all T tokens for that block and has valid data. Token coherence avoids starvation by issuing a persistent request when a core detects potential starvation. In CMP systems, it uses a distributed arbitration scheme for persistent requests, which are issued after a single retry to optimize the access to contended blocks. Particularly, on every cache miss, the requesting core broadcasts requests to all other tiles. In case of a write miss, they have to answer with all tokens that they have. The data block is sent along with the owner token. When the requester receives all tokens the block can be accessed. On the other hand, just one token is required upon a read miss. The request is broadcast to all other tiles, and only those that have more than one token (commonly the one that has the owner token) answer with a token and a copy of the requested block. Figure 2.11 shows an example of how Token solves a cache-to-cache transfer miss. Requests are broadcast to all tiles (1 GetX). The only tile with tokens for that block is M, which responds by sending the data and all the tokens (2 Data).

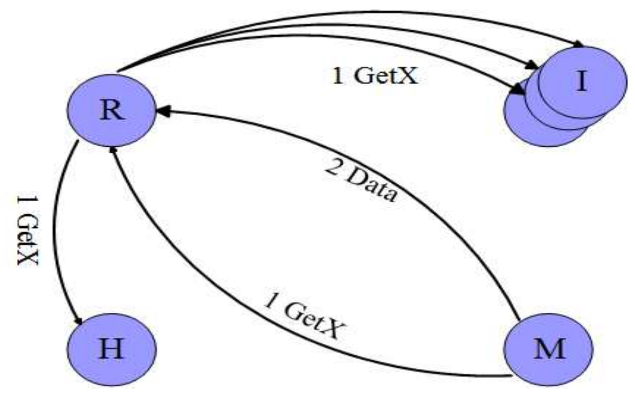


Figure 2.11: Cache to Cache miss in Token coherence protocol

We can see that this protocol avoids indirection since only two hops are introduced in the critical path of cache misses. However, as happens in Hammer, this protocol also has the drawback of broadcasting requests to all tiles on every cache miss, which results in high network traffic and, consequently, power consumption at the interconnect.

2.5.4 Directory-based protocol

One of the most widely used cache coherence protocol [85] [89] in shared memory multiprocessors was directory based coherence protocol. A number of conventional multiprocessors that employed directory protocols are the Standford DASH [91] and FLASH [92] multiprocessors, the SGI Origin 2000/3000 [93], and the AlphaServer GS320 [94]. Currently, a number of Chip Multiprocessors, such as Piranha [95] or Sun UltraSPARC T2 [96], as well employs directory protocols to maintain cache coherence. In directory based protocols, the serialization location is also the home core of the block, which is similar to the hammer protocol. In comparison to hammer, the directory based protocols refrain from transmitting the requests by maintaining details about the state of every individual block within the private caches. This data is known as directory information (therefore known as directory-based protocols). In an effort to speed up cache misses, this directory details are not kept in main memory. Rather, it is often stored on-chip with the home tile of each and every block. The directory-based protocol that we have implemented for CMPs is similar to the intra-chip coherence protocol used in Piranha. Essentially, the directory data is comprised of a full-map (or bit-vector) sharing code that is utilized for tracking the sharers of the block. Sharing code permits protocol in terms of sending invalidation messages to caches that are currently sharing block, and so remove unnecessarily identified coherent messages. In directory-based protocols, where O-state is for '*an owner field*', referring to owner tile gets added to directory information meant for every block. Owner field permits protocols in terms of detecting tile that needs to offer block meant under varied sharers. Thus, requests are forwarded to tile. Application of directory information permits protocol for reducing adequate network traffic as against Hammer as well as Token. Over each cache missed in application is marked in directory protocol [97] [98], where core reason to miss sends request to home tile, an aspect that serializes for all kinds of requests issued in terms of same block. Every home tile comprises of on-chip directory cache that is responsible for storing, sharing and further owning data for blocks as it manages. This cache gets implied for blocks that never hold any copy in shared cache. Moreover, tags' part of the shared cache comprises field for storing all the shared data meant for those blocks with a valid entry in cache. As home tile decides about the request process, it gains access towards directory data and further performs apt kinds of coherent actions. These actions comprise forwarding request to owner tile, and further lay interest in invalidating all the block copies for an instant where write gets missed. Whenever, the tile obtains a forwarding request, it sends the data towards the consumer when it is already available or, in other instance, the request need to hold back until the data will be available. Similar to Hammer, each and every tiles must reply to the invalidation messages using an acknowledgement message to the consumer. Since, the acknowledgement messages are obtained by the requester, it is often essential to update the consumer regarding the number of acknowledgements that it must obtain prior to accessing the requested data block. Within the implementation that we use in this thesis, this information is sent from the home tile, which has knowledge of the total number of invalidation messages released, to the requester in addition to forwarding as well as data messages. Whenever the consumer obtains all acknowledgements and the data block, it unblocks the home tile in order to permit it to process additional requests for that block. Figure 2.12 presents an illustration of how directory resolves a cache-to-cache transfer miss. The request is forward towards the home tile, wherein the directory data is preserved (1 GetX). After that, the home tile forwards the request towards the source of the block, which is extracted from the directory data (2 Fwd). Whenever the data forwarded by the provider arrives towards the requester (3 Data), the miss is considered solved and the home tile must be unblocked (4 Unbl). As we can see, although this protocol presents indirection to resolve cache misses (about three hops within the critical path of the miss), a small number of coherence messages are involved to resolve them, which ultimately translates into savings in network traffic and less power consumption. This attribute makes the directory protocol the utmost scalable approach.

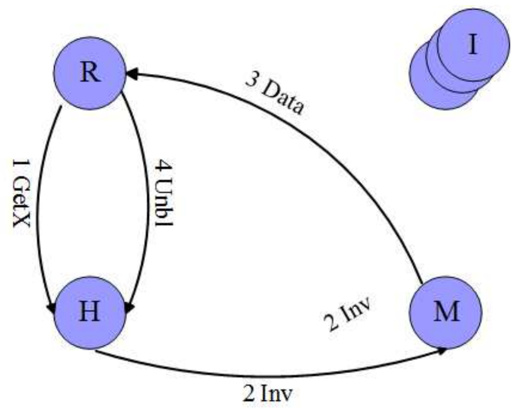


Figure 2.12: Cache to Cache transition in Directory based coherence protocol

The cache coherence protocols explained earlier are summarized in Table 2.1. It is not feasible to employ conventional snooping-based protocols for scalable point-to-point networks. Hammer can work over scalable point-to-point networks but at the expense of broadcasting requests to all cores and introducing indirection in the critical path of cache misses. Tokens on the other hand, evade the indirection but yet send requests to all cores on every cache miss, which, in turn, influences the consumption of power and network traffic. Contradictorily, directory merely sends requests to the core that must obtain them; it however, initiates indirection, which influences the execution time of the applications.

Table 2.1: Summary of coherence protocols

	Network	Requests	Indirection
Snooping	shared interconnect	To all cores	No
Hammer	Point-to-point	To all cores	Yes
Token	Point-to-point	To all cores	No
Directory	Point-to-point	Only to necessary	Yes

In our work in this thesis, we have used modified directory-based coherence protocols that circumvents both broadcasting messages to each of the cores and the indirection to the owner core for majority of cache misses.

2.6 Summary

In this chapter, we reviewed the existing research done to optimize cache management schemes for uni-cores and multicore processors. We focused on some fundamental work followed by description of existing cache coherence techniques for multiprocessors.

Chapter 3

Evaluation Methodology

This chapter describes the experimental framework and the benchmarks used in this thesis.

Chapter 3

Evaluation Methodology

3.1 Introduction

The Experimental frame work and the methodology employed in this thesis is described in the current chapter. First, a flexible and detailed cache-coherent distributed shared memory system prototype that comprises of L1 caches, L2 caches, main memory and interconnection network is implemented. Then details of the timing simulation tool are presented followed by the discussion of the power and area estimation tools.

The remainder of the chapter is structured as follows: Section 3.2 presents the details of the simulation tools used for the performance evaluations carried out in the current work. Section 3.3 and Section 3.4 discusses the interconnection and power estimation tools used for calculating the improvements for the proposed schemes. At last, the discussions on the choice of application programs (benchmarks) and their descriptions are presented in Section 3.5.

3.2 Experimental Frame work

We employed an in-order processor model with the emphasis on the average raw memory latency encountered by each memory request to provide a much better illustration of the memory system behavior.

3.2.1 Simulation Tools (Simulation Setup)

We have used Simics from Virtutech, which is a full system simulator [99] that has the ability of simulating an entire computing system, including processors, caches and memories, graphics and networking cards, hard disks, and many types of removable media. This kind of flexibility allows the simulation of many different hardware architectures and the ability to boot a variety of different operating systems. Better yet, the ability to boot these operating systems means that there are a variety of benchmarking suites available to test system optimizations. Simics provides a built-in cache system called g-cache that allows individual cache modules to be attached to a processor. Using these cache modules it is possible to build up a model of the entire cache system, including simulating accesses to main memory. The g-cache implementation even provides support for a built-in coherency protocol called MESI, which is used in a many of Intel’s microprocessors. While this implementation of MESI is specifically intended for cache systems utilizing write-through L1 caches and write-back L2 caches, it can be modified to work for other configurations. MESI, which represents Modified Exclusive Shared Invalid, provides a method for indicating the status of lines within the cache. Limiting the number of states to four requires that only two bits be added to each line in the cache, resulting in a relatively small storage overhead.

At present, Simics backs prototypes for the following architectures: UltraSPARC, Alpha, x86, x86-64 (Hammer), PowerPC, IPF (Itanium), MIPS and ARM. In addition to the ability of simulating target architectures, Simics easily allows the inclusion of extensions or modules in order to extend its functionalities.

3.2.2 Detailed Cache Simulator

We employed a modular simulation infrastructure GEMS (General Execution-driven Multiprocessor Simulator) that decouples both stimulation functionality and timing so as to develop a simulation tool-set that endorses both full-system and timing simulation [100]. We utilized Simics [99], a full-system functional simulator, as the basis on which different timing simulation units could be loaded dynamically. We control both the efficacy and robustness of a functional simulator by decoupling functionality and timing simulation in GEMS. The employment of the modular design offers the adaptability to simulate different system modules in varied levels of details. GEMS include a group of modules executed in C++ that plug into Simics and add timing capacities to the simulator. GEMS provide offers varied modules for designing different facets of architecture.

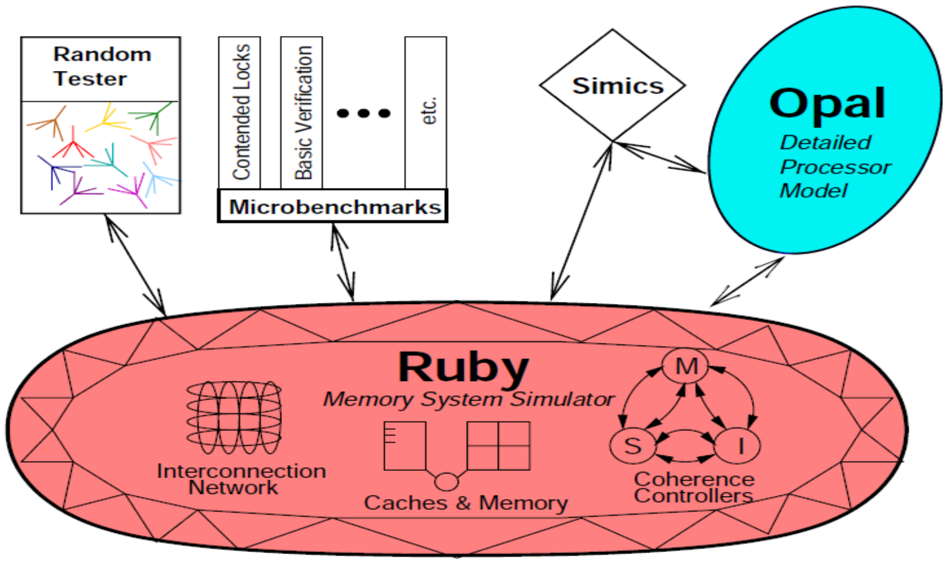


Figure. 3.1: A block diagram of GEMS Structure: Ruby, detailed memory simulator can be driven by one of four memory system request generators

The heart of GEMS is the Ruby memory system simulator. As illustrated in Figure 3.1, GEMS provides multiple drivers that can serve as a source of memory operation requests to Ruby:

1) **Random tester module**: The most basic driver of Ruby is a random testing unit designed to stress test the corner cases of a given memory organization. It makes use of false sharing as well as action/check pairs to identify several possible memory system as well as coherence issues in addition to race problems [28]. A number of capabilities are found in Ruby that can debug the modelled system along with deadlock identification as well as protocol tracing.

2) **Micro-benchmark module**: This driver allows several micro-benchmarks with a common interface. The feature work extremely well for fundamental timing validation, in addition to comprehensive performance evaluation of certain conditions (e.g., lock contention or widely-share data).

3) **Simics**: This driver makes use of Simics functional simulator to effectively approximate a reliable in-order processor without pipeline stalls. Simics sends each and every load, store, and instruction fetch requests to Ruby, which carries out the first stage cache access to find out if the operation hits or misses within the first level cache. Upon the cache hit, Simics may keep executing instructions, switching between processors within a multiple processor setting. On a cache miss, Ruby stalls Simics' request originating from issuing processor, and thereafter simulates the cache miss. Every individual processor could have only a single miss outstanding, however contention along with other timing affects among the cpu cores will decide when the request finishes. By governing the timing related to when Simics advances, Ruby decides the timing-dependent functional simulation in Simics (e.g., to identify which processor subsequently receives a memory block).

4) **Opal**: This powerful driver models a dynamically-scheduled SPARC v9 processor and certainly utilizes Simics to verify its functional correctness.

   The initial pair of drivers belong to a stand-alone executable that is separate from Simics or any real simulated program. Moreover, Ruby is particularly developed to assist various other drivers other than four already mentioned by means of well-defined interface. GEMS' modular layout offers considerable simulator configuration flexibility. For example, these memory system simulator is separate from our processor simulator. GEMS additionally provides flexibility in specifying several cache coherence protocols that can be simulated by our timing simulator. It divides the protocol-dependent information from the protocol-independent system components as well as techniques. To facilitate specifying different protocols and systems, it provides the protocol specification language SLICC which we have used for implementing the proposed cache management techniques. The two main simulation modules are Ruby and Opal.

5) **Ruby:** Ruby has been identified as timing simulator for multiprocessors memory system which includes caches, controllers of cache and memory, interconnection network, and main memory banks. Ruby comprises hard-coded timing simulation in relation with components that remain largely independent over cache-coherent protocol (like, interconnecting network) added by the capability to describe protocol-dependent elements (as cache controllers) in terms of domain-specific language, SLICC (Specification Language for Implementing Cache Coherence). Ruby module is realized using C++ and further uses queue-driven model for simulating timing. Message buffers of different latencies and bandwidth are used for communication in between various components, in addition the components at the receiving end of the buffer are scheduled to get up over next message, which is available for reading from the buffer. However, there are many buffers that are used under strict first-in-first-out (or the FIFO) manner, whereby the buffers are never liable to remain restricted towards FIFO behavior. The simulation proceeds by invoking the wakeup method for the next scheduled event on the event queue. Thus, simulation remains identical, in case all the components get woken up in every cycle; so that event queue can get optimized for avoiding unnecessary processing in every cycle.

3.2.3 Protocol-Independent Components

The message buffer, cache arrays, memory arrays and assorted glue logic are the protocol independent components of ruby. However, a pair of components that deserves discussion are definitely the caches as well as the interconnection network.

Caches: Ruby module permits to implement a complete cache hierarchy associated with each single core in addition to the shared caches employed in the CMPs along with other hierarchical coherence system. Cache attributes which can include size and associativity, are considered as the configuration parameters.

Interconnection Network: The interconnection network is the unified communication substrate used to communicate between cache and memory controllers. A single monolithic interconnection network model is used to simulate all communication, even between controllers that would be on the same chip in a simulated CMP system. As such, all intra-chip and inter-chip communication is handled as part of interconnect, although each individual link can have different latency and bandwidth parameters. This design provides sufficient flexibility to simulate the timing of almost any kind of system. A controller communicates by sending messages to other controllers. Ruby's interconnection network models the timing of the messages as they traverse the system. Messages sent to multiple destinations (such as a broadcast) use traffic-efficient multicast-based routing to fan out the request to the various destinations. Ruby models a point-to-point switched interconnection network that can be configured similarly to interconnection networks in current high-end multiprocessor systems, including both directory-based and snooping-based systems.

For simulating systems based on directory protocols, Ruby supports three non-ordered networks: a simplified full connected point-to-point network, a dynamically-routed 2D-torus interconnect inspired, and a flexible user-defined network interface. The first two networks are automatically generated using certain simulator configuration parameters, while the third creates an arbitrary network by reading a user-defined configuration file. This file-specified network can create complicated networks such as a CMP-DNUCA network. For snooping-based systems, Ruby has two totally-ordered networks: a crossbar network and a hierarchical switch network. Both ordered networks use a hierarchy of one or more switches to create a total order of coherence requests at the network's root. This total order is enough for many broadcast-based snooping protocols, but it requires that the specific cache-coherence protocol does not rely on stronger timing properties provided by the more traditional bus-based interconnect. The topology of interconnect is specified by a set of links between switches, and the actual routing tables are re-calculated for each execution, allowing for additional topologies to be easily added to the system. The interconnect models virtual networks for different types and classes of messages, and it allows dynamic routing to be enabled or disabled on a per-virtual-network basis (to provide point-to-point order if required). Each link of interconnect has limited bandwidth, but interconnects does not model the details of the physical or link-level layer. By default, infinite network buffering is assumed at the switches, but Ruby also supports finite buffering in certain networks. Although, Ruby's interconnect model is sufficient for coherence protocol and memory hierarchy research, but it allows integration of more detailed interconnection network for research focusing on low-level interconnection network issues.

3.2.4 Specification Language for Implementing Cache Coherence (SLICC)

A domain-specific language is included under Ruby to state cache coherence protocols referred to as SLICC (Specification Language for Implementing Cache Coherence).SLICC permits the effortless development of varied cache coherence protocols and it has been employed to implement the protocols assessed in the current study. It relies on notion of stating distinct controller state machines that represent system elements like cache controllers and directory controllers. Each controller is theoretically a per-memory-block state machine, which comprises of:

- **States**: group of probable states for each cache block,
- **Events**: conditions that activate state changes, like message arrivals,
- **Transitions:** the cross-result of states and events (relying on the state and event, a transition executes an atomic series of activities and modifies the block to a new state)
- **Actions:** the particular operation executed during a transition.

For instance, the SLICC code may specify a “Shared” state that permits read-only access for a block in a cache.

3.3 Interconnection Network

3.3.1 GEMS Interconnection Network

Our initial simulation uses GEMS’s [100] network model for interconnect and switch contention prototype. It uses virtual cut-through switching for transferring cache messages through interconnects. The network link width is 16 bytes and so is the flit size. The data and command communications are executed by messages of three varied sizes (8 bytes, 16 bytes, and 72 bytes). 8-byte and 72-byte messages are used by L2S and Victim Migration. The network link is shared at an 8B granularity; this indicates two 8B messages (or one 8B message and part of a 16B or 72B message) can be sent at the same time, presuming that both the messages are to be sent. We worked with a buffer size of 3 and a message is allowed to go through a link/switch in cases where there is a free buffer entry on the other side. The link is occupied across most of the cycles required (determined by message size, link-width, and link latency) to forward a message from one particular side to the other side. Messages are forwarded in first come first serve (FCFS) manner. L2S makes use of 2 virtual channels (one for request and one more for response messages) in each direction. Messages swapped between L1s as well as L2s are treated as on-chip traffic and messages communicated between L2S and memory controller are treated as off-chip traffic.

3.3.2 Garnet Network / Orion

With increasing core counts, the on-chip network becomes an integral part of future chip multiprocessor (CMP) systems. Future CMPs, with dozens to hundreds of nodes, will require a scalable and efficient on-chip communication fabric. There are several ways in which on-chip communication can affect higher-level system design. Contention delay in the network, as a result of constrained bandwidth, impacts system message arrivals. In multi-threaded applications, spin locks and other synchronization mechanisms magnify small timing variations into very different execution paths. Network protocols also impact the ordering of messages. A different order of message arrival can impact the memory system behavior substantially. Especially for cache coherence protocols, protocol level deadlocks are carefully avoided by designing networks that obey specific ordering properties among various protocol messages. The manner in which the ordering is implemented in the network leads to different messages seeing different latencies and again impacts message arrivals. Communication affects not only performance, but can also be a significant consumer of system power. Not only do network characteristics impact system-level behavior, the memory system also impacts network design to a huge extent. Co-designing interconnects and the memory system provides the network with realistic traffic patterns and leads to better retuning of network characteristics

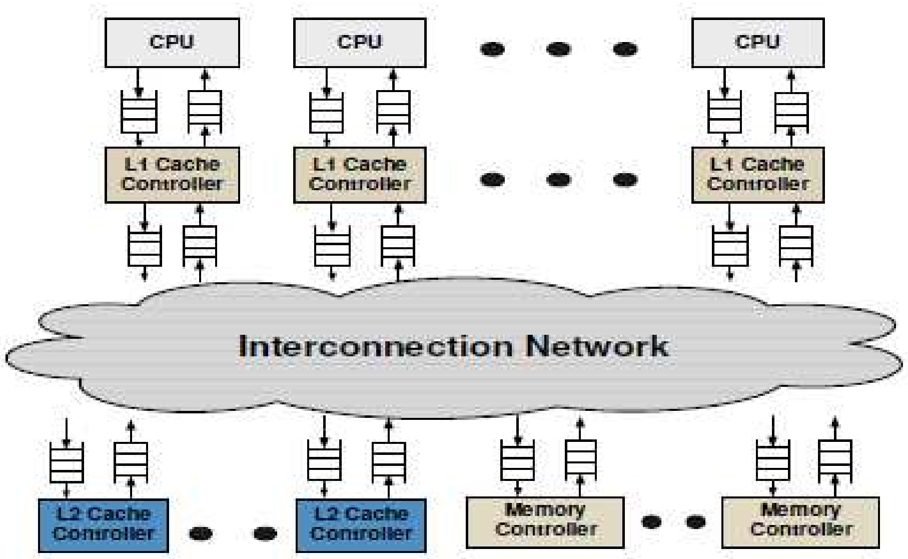


Figure 3.2: Interconnection network on chip

System-level knowledge can highlight which metric (delay/throughput/power) is more important. The inter-connect also needs to be aware of the specific ordering requirements of higher levels of design. Figure 3.2 shows how various components of a CMP system are coupled together. The inter-connection network is the communication backbone of the memory system. Thus, interconnection network details can no longer be ignored during memory system design. To study the combined effect of system and interconnect design, we require a simulation infrastructure that models these aspects to a sufficient degree of detail. In most cases, it is difficult to implement a detailed and accurate model that is fast enough to run realistic workloads. Adding detailed features increases the simulation overhead and slows it down. However, there are some platforms that carefully trade off accuracy and performance to sufficiently abstract important system characteristics while still having reasonable speed of simulation on realistic workloads. One such platform is the GEMS full-system simulation platform. It does a good job in capturing the detailed aspects of the processing cores, cache hierarchy, cache coherence, and memory controllers. This has led to widespread use of GEMS in the computer architecture research community. There has been a huge body of work that has used GEMS for validating research ideas. One limitation of GEMS, however, is its approximate interconnect model. The interconnection substrate in GEMS serves as a communication fabric between various cache and memory controllers. The model is basically a set of links and nodes that can be configured for various topologies with each link having a particular latency and bandwidth. For a message to traverse the network, it goes hop by hop towards the destination, stalling when there is contention for link bandwidth. GEMS does not model a detailed router or a network interface. By not modeling a detailed router micro architecture, GEMS ignores buffer contention, switch and Virtual Channel (VC) arbitration, realistic link contention and pipeline bubbles. The GEMS interconnect model also assumes perfect hardware multicast support in the routers. In on-chip network designs, supporting fast and low power hardware multicast is a challenge. These and other limitations in the interconnect model can significantly affect the results reported by the current GEMS implementation. Trace driven techniques also do not capture program variability that a full-system evaluation can. In the light of the above issues, we have integrated GARNET, which is a detailed timing model of a state-of-the-art interconnection network, modeled in detail up to the microarchitecture level. A classical five-stage pipelined router with virtual channel flow control is implemented. Such a router is typically used for high-bandwidth on-chip networks.

3.3.3 Base GARNET model design

Modern on-chip network designs use a modular packet-switched fabric in which network channels are shared over multiple packet flows. We used a classic five-stage virtual channel router [101]. The router can have any number of input and output ports depending on the topology and configuration. The major components, which constitute a router, are the input buffers, route computation logic, VC allocator, switch allocator and crossbar switch. A five-stage router pipeline was selected to adhere to high clock frequency network designs. Every VC has its own private buffer. The routing is dimension-ordered. Since research in providing hardware multicast support is still in progress and state-of-the art on-chip networks do not have such support, we do not model it inside the routers. A head it, on arriving at an input port, first gets decoded and gets buffered according to its input VC in the buffer write (BW) pipeline stage. In the same cycle, a request is sent to the Route Computation (RC) unit simultaneously, and the output port for this packet is calculated. The header then arbitrates for a VC corresponding to its output port in the VC allocation (VA) stage. Upon successful allocation of an output VC, it proceeds to the Switch Allocation (SA) stage where it arbitrates for the switch input and output ports. On winning the switch, then it moves to the switch traversal (ST) stage, where it traverses the crossbar. This is followed by Link Traversal (LT) to travel to the next node. Body and tail its follow a similar pipeline except that they do not go through RC and VA stages, instead inheriting the VC allocated by the head it. The tail it on leaving the router, deallocates the VC reserved by the packet. Router micro architectural components: Keeping in mind on-chip area and energy considerations, single-ported buffers and a single shared port into the crossbar from each input were designed. Separable VC and switch allocators were modeled. This was done because these designs are fast and of low complexity, while still providing reasonable throughput, making them suitable for the high clock frequencies and tight area budgets of on-chip networks.

The individual allocators are round-robin in nature. Interactions between memory system and garnet as shown in Figure 3.1. The interconnection network acts as the communication backbone for the entire memory system on a CMP. The various L1 and L2 cache controllers and memory controllers communicate with each other using the interconnection network. Note that we are talking about a shared L2 system here. The network interface acts as the interface between various modules and the network. On a load/store, the processor looks in the L1 cache. On a L1 cache miss, the L1 cache controller places the request in the request buffer. The network interface takes the message and breaks it into network-level units (its) and routes it to the appropriate destinations which might be a set of L1 controllers as well as L2 controllers. The destination network interfaces combine this into the original request and pass it on to the controllers. The responses use the network in a similar manner for communication. Some of these messages might be on the critical path of memory accesses. A poor network design can degrade the performance of the memory system and also the overall system performance. Thus, it is very important to architect the interconnection network efficiently.

3.4 Energy Model

3.4.1 CACTI

CACTI (Cache Access and Cycle Time Information) [102] provides an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, one can get to know trade-offs between time, power, and area. CACTI is continually being upgraded due to the incessant improvements in semiconductor technologies. Particularly, we employ the version 5.3 for the results presented in this thesis. We are mainly interested in getting the access latencies and area requirements of both cache and directory structures that are necessary for implementing our ideas. In this study, we assume that the length of the physical address is 40 bits as, for example, in the Sun UltraSPARC T2 architecture [96]. This length is used to calculate the bits required to store the tag field for each cache. Moreover, we also assume a 45nm process technology, and the other parameters shown in the following section.

3.4.2 Energy calculation

This thesis also evaluates the energy consumed by the NUCA cache and the off-chip memory. To do so, we used a similar energy model to that adopted by Bardine et al. [103]. This allowed us to also consider the total energy dissipated by the NUCA cache and the additional energy required to access the off-chip memory. The energy consumed by the memory system is computed as follows:

$$Etotal = Estatic + Edynamic$$

$$Estatic = ES\_noc + ES\_banks + ES\_mechanism$$

$$Edynamic = ED\_noc + ED\_banks + ED\_mechanism + Eoff{-}chip$$

We used models provided by CACTI to evaluate static energy consumed by the memory structures (ES_banks and ES_mechanism). CACTI has been used to evaluate dynamic energy consumption as well, but GEMS support is required in this case to ascertain the dynamic behavior in the applications (ED_banks and ED_mechanism). GEMS also contains an integrated power model based on Orion [104] that we used to evaluate the static and dynamic power consumed by the on-chip network (ES_noc and ED_noc). Note that the extra messages introduced by the mechanism that is being evaluated into the on-chip network are accurately modeled by the simulator. The energy dissipated by the off-chip memory (Eoff−chip) was determined using the Micron System Power Calculator [105] assuming a modern DDR3 system (4GB, Vdd: 1.5v, 333 MHz). Our evaluation of the off-chip memory focused on the energy dissipated during active cycles and isolated this from the background energy. This study shows that the average energy of each access is 550 pJ. As an energy metric we used the energy consumed per memory access. This is based on the energy per instruction (EPI) metric which is commonly used for analyzing the energy consumed by the whole processor. This metric works independently of the amount of time required to process an instruction and is ideal for throughput performance.

3.5 Workload Description

The aim of this section is to choose a benchmark suite that can be used to design the next generation of processors. In this section, we first present the requirements for such a suite. We then discuss how the existing benchmarks fail to meet these requirements.

We have the following five requirements for a benchmark suite:

Multithreaded Applications: Shared-memory CMPs are already ubiquitous. The trend for future processors is to deliver large performance improvements through increasing core counts on CMPs while only providing modest serial performance improvements. Consequently, applications that require additional processing power will need to be parallel.

Emerging Workloads: Rapidly increasing processing power is enabling a new class of applications whose computational requirements were beyond the capabilities of the earlier generation of processors. Such applications are significantly different from earlier applications.

Diverse Workloads: Applications are increasingly diverse, run on a variety of platforms and accommodate different usage models. They include both interactive applications like computer games, offline applications like data mining program and programs with different parallelization models. Specialized collections of benchmarks can be used to study some of these areas in more detail, but decisions about general-purpose processors should be based on a diverse set of applications. While a truly representative suite is impossible to create, reasonable effort should be made to maximize the diversity of the program selection. The number of benchmarks must be large enough to capture a sufficient amount of characteristics of the target application space.

Employ State-of-Art Techniques: A number of application areas have changed dramatically over the last decade and use very different algorithms and techniques. Visual applications for example have started to increasingly integrate physics simulations to generate more realistic animations. A benchmark should not only represent emerging applications but also use state-of-art techniques.

Support Research: A benchmark suite intended for research has additional requirements compared to one used for benchmarking real machines alone. Benchmark suites intended for research usually go beyond pure scoring systems and provide infrastructure to instrument, manipulate, and perform detailed simulations of the included programs in an efficient manner.

3.5.1 Limitations of Existing Benchmark Suites

In the remaining part of this section we analyze how existing benchmark suites fall short of the presented requirements and must thus be considered unsuitable for evaluating CMP performance.

SPEC CPU2006 and OMP2001: SPEC CPU2006 and SPEC OMP2001 [106] are two of the largest and most significant collections of benchmarks. They provide a snapshot of current scientific and engineering applications. Computer architecture research, however, commonly focuses on the near future and should thus also consider emerging applications. Workloads such as systems programs and parallelization models which employ the producer-consumer model are not included. SPEC CPU2006 is furthermore a suite of serial programs that is not intended for studies of parallel machines.

SPLASH-2: This is a suite composed of multithreaded applications [107] and hence seems to be an ideal candidate to measure performance of CMPs. However, its program collection is skewed towards HPC and graphics programs. It thus does not include parallelization models such as the pipeline model which are used in other application areas. SPLASH- 2 should furthermore not be considered state-of-art anymore. Barnes for example implements the Barnes-Hut algorithm for N-body simulation. For galaxy simulations it has largely been superseded by the TreeSPH method, which can also account for mass such as dark matter which is not concentrated in bodies. However, even for pure N-body simulation barnes must be considered outdated. In 1995 Xu proposed a hybrid algorithm which combines the hierarchical tree algorithm and the Fourier-based Particle-Mesh (PM) method to the superior TreePM method. Our analysis shows that similar issues exist for a number of other applications of the suite including raytrace and radiosity.

Other Benchmark Suites Apart from all the major types of benchmark suites, there are various smaller collections of workloads that are in general designed in order to research over determined program area and therefore remain limited towards single application domain. This is the reason that such aspects remain inclusive of smaller application sets against diversified benchmark suite, offered typically. Such limitations are not applicable for scientific research, which do not restrict the application domain. Such type of benchmark suites can be noted as ALPBench, MineBench, MediaBench, BioParallel and Physics-Bench. As they follow diversified approaches, we will not discuss such suites in detail.

3.5.2 Multi-threaded Benchmarks

One of the goals of the PARSEC suite was to assemble a program [108] selection that is large and diverse enough to be sufficiently representative for scientific studies. It consists of 9 applications and 3 kernels which were chosen from a wide range of application domains. PARSEC workloads were selected to include different combinations of parallel models, machine requirements and runtime behaviors. All benchmarks are written in C/C++ because of the continuing popularity of these languages in the near future. PARSEC meets all the requirements outlined in Section 3.4.3:

- All the applications are parallelized
- PARSEC benchmark suite never gets skewed for HPC programs that appear in abundance, yet represent just a niche. It lays importance over emerging workloads.
- Diversified workloads are selected from various areas like media processing, computational, computer vision, enterprise servers, finance and animation physics. PARSEC appears more diverse against SPLASH-2.
- Every application represents state-of-art in respective areas.
- PARSEC supports computer architecture research in a number of ways. The most important one is that for each workload six input sets with different properties are defined

(Section 3.1). The characteristics of the included workloads differ substantially from SPLASH-2 [6].

Recent technology trends such as the emergence of CMPs and the growth of world data seem to have a strong impact on workload behavior.

3.5.2.1 Input Sets

PARSEC defines six input sets for each benchmark:

TEST: A very small input set to test the basic functionality of the program.

SIMDEV: A very small input set which guarantees basic program behavior similar to the real behavior, intended for simulator test and development.

SIMSMALL, SIMMEDIUM and SIMLARGE: Input sets of different sizes suitable for simulations.

NATIVE: A large input set intended for native execution.

TEST and **SIMDEV** are merely intended for testing and development and should not be used for scientific studies. The three simulator inputs for studies vary in size, but the general trend is that larger input sets contain bigger working sets and more parallelism. Finally, the native input set is intended for performance measurements on real machines and exceeds the computational demands which are generally considered feasible for simulation by orders of magnitude.

3.5.2.2 Workloads

The following workloads are part of the PARSEC suite:

BLACKSCHOLES: This application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). There is no closed-form expression for the Black- Scholes equation and as such it must be computed numerically.

BODYTRACK: This computer vision application is an Intel RMS workload which tracks a human body with multiple cameras through an image sequence. This benchmark was included due to the increasing significance of computer vision algorithms in areas such as video surveillance, character animation and computer interfaces.

CANNEL: This kernel was developed by Princeton University. It uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design. Canneal uses fine-grained parallelism with a lock-free algorithm and a very aggressive synchronization strategy that is based on data race recovery instead of avoidance.

DEDUP: This kernel was developed by Princeton University. It compresses a data stream with a combination of global and local compression that is called ’deduplication’. The kernel uses a pipelined programming model to mimic real-world implementations. The reason for the inclusion of this kernel is that deduplication has become a mainstream method for new generation backup storage systems.

FACESIM: This Intel RMS application was originally developed by Stanford University. It computes a visually realistic animation of the modeled face by simulating the underlying physics. The workload was included in the benchmark suite because an increasing number of animations employ physical simulation to create more realistic effects.

FERRET: This application is based on the Ferret toolkit which is used for content-based similarity search. It was developed by Princeton University. The reason for the inclusion in the benchmark suite is that it represents emerging next-generation search engines for non-text document data types. In the benchmark, we have configured the Ferret toolkit for image similarity search. Ferret is parallelized using the pipeline model.

FLUDANIMATE: This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. It was included in the PARSEC benchmark suite because of the increasing significance of physics simulations for animations.

FREQMINE: This application employs an array-based version of the FP-growth (Frequent Pattern-growth) method for Frequent Item set Mining (FIMI). It is an Intel RMS benchmark which was originally developed by Concordia University. Freqmine was included in the PARSEC benchmark suite because of the increasing use of data mining techniques.

STREAMCLUSTER: This RMS kernel was developed by Princeton University and solves the online clustering problem. Streamcluster was included in the PARSEC benchmark suite because of the importance of data mining algorithms and the prevalence of problems with streaming characteristics.

SWAPTIONS: The application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. Swaptions employs Monte Carlo (MC) simulation to compute the prices.

VIPS: This application is based on the VASARI Image Processing System (VIPS) which was originally developed through several projects funded by European Union (EU) grants. The benchmark version is derived from a print on demand service that is offered at the National Gallery of London, which is also the current maintainer of the system. The benchmark includes fundamental image operations such as an affine transformation and a convolution.

X264: This application is an H.264/AVC (Advanced Video Coding) video encoder. H.264 describes the lossy compression of a video stream and is also part of ISO/IEC MPEG-4. The flexibility and wide range of application of the H.264 standard and its ubiquity in next-generation video systems are the reasons for the inclusion of x264 in the PARSEC benchmark suite.

3.5.2.3 Characterization

We are interested in the following benchmark characteristics:

Parallelization: PARSEC benchmarks use different parallel models which have to be analyzed in order to know whether the programs can scale well enough for the analysis of CMPs of a certain size.

Working sets and locality: Knowledge of the cache requirements of a workload are necessary to identify benchmarks suitable for the study of CMP memory hierarchies.

Communication-to-computation ratio and sharing: The communication patterns of a program determine the potential impact of private caches and the on-chip network on performance.

Off-chip traffic: The off-chip traffic requirements of a program are important to understand how off-chip bandwidth limitations of a CMP can affect performance.

In Table 3.1 we summarize the important characteristics of the identified working sets. Most workloads exhibit well defined working sets with clearly identifiable points of inflection. Compared to SPLASH-2, PARSEC working sets are significantly larger and can reach hundreds of megabytes such as in the cases of canneal and freqmine.

Two types of workloads can be distinguished:

The first group contains benchmarks such as bodytrack and swaptions which have working sets no larger than 16 MB. These workloads have a limited need for caches with a bigger capacity, and the latest generation of CMPs often already has caches sufficiently large to accommodate most of their working sets.

The second group of workloads is composed of the benchmarks canneal, ferret, facesim, fluidanimate and freqmine. These programs have very large working sets of sizes 65 MB and more, and even with a relatively constrained input set such as simlarge, their working sets can reach hundreds of megabytes.

Table 3.1: Benchmark characteristics

Program	Application Domain	Parallelization		Working Set	Date Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
Canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Engineering Storage	pipeline	medium	unbounded	high	high
Facesim	Animation	data-parallel	coarse	large	low	medium
Ferret	Similarity search	pipeline	medium	unbounded	high	high
Fluidanimate	Animation	data-parallel	fine	large	low	medium
Freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
Streamcluster	Data Mining	data-parallel	medium	medium	low	medium
Swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
Vips	Media processing	data-parallel	coarse	medium	low	medium
X264	Media processing	pipeline	coarse	medium	high	high

Furthermore, the requirement of those workloads for cache capacity is almost voracious and rises with the extent of data processed by them. The Table 3.1 outlines the approximations for the biggest working set of every PARSEC workload for the native input group. In many instances, they are remarkably huge and can even touch gigabytes. These huge working groups are commonly the result of an algorithm that functions and is based on huge amounts of input data that is gathered. Canneal, dedup, ferret and freqmine are programs with unbounded working groups.

3.5.3 Multi-programmed workloads

We also evaluated our proposed schemes with multi-programmed workloads, which comprise of many application instances running at the same time using different subsets of the cores available on chip. As it is anticipated that several core architectures will also be used for throughput computing and multi-programmed workloads have varied protocol requirements as compared to parallel applications, they also make an interesting scenario for the assessment undertaken in this work.

3.6 Summary

To summarize, Simics full sytem simulator along with GEMS enables us to model cache memory, interconnects and off-chip memory with moderate accuracy and power is measure by CACTI. We have carefully chosen applications from diverse domains. The multiprogramming workloads that we simulate have applications with varying memory intensive and non-intensive properties. We consider an eight core chip multiprocessor in all our experiments. And finally, we have evaluated our proposed schemes by executing these diverse memory intensive applications on our baseline architecture with proposed cache management schemes.

Chapter 4

Adaptive Block Pinning: A Novel Shared Cache Partitioning Scheme for CMP

This chapter presents an Adaptive Block Pinning Scheme, which is a Novel Shared Cache Management Scheme for CMPs to reduce miss rate.

Chapter 4

Adaptive Block Pinning: A Novel Shared Cache Partitioning for CMP

4.1 Introduction

Traditionally, multi-processor systems have been designed by interconnecting multiple uniprocessors and DRAM modules. In comparison to uniprocessors, a multi-chip system is capable of delivering computing power that is several magnitudes higher. However, the design and performance of the memory system for multi-chip system directly affects the overall system performance. Below are three different alternatives, each differing in the way they store and access data. Figure 4.1, shows a physically centralized memory shared by all of the processors, interconnected through a shared bus. While this approach is simple, it can only be applied when the number of processors in the system is small.

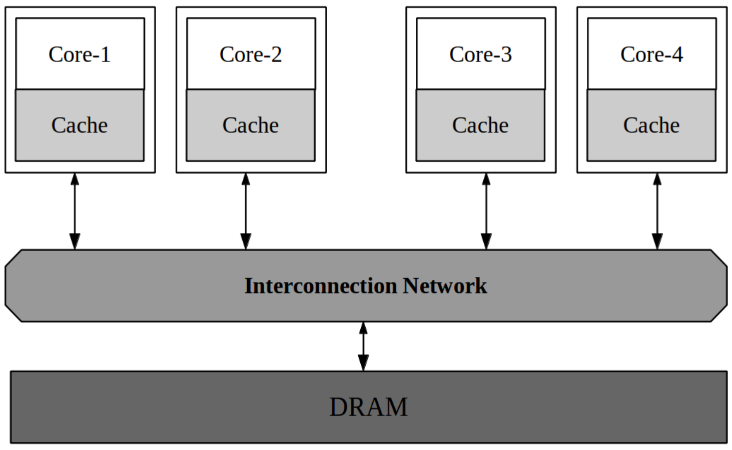


Figure 4.1: Multi-processors with physically centralized memory

Large multi-chip systems generally have hundreds of processors and the bandwidth of a centralized memory system does not scale with the processor count. In these large multi-chip systems, physical memory is typically distributed across the system, with a portion of the memory co-located with each processor. A communication protocol is used to manage the exchange of shared data between different processors. Two such approaches are illustrated in Figure 4.2 and Figure 4.3. Traditionally, designers have taken two approaches for implementing a physically distributed memory system: message passing and distributed shared memory.

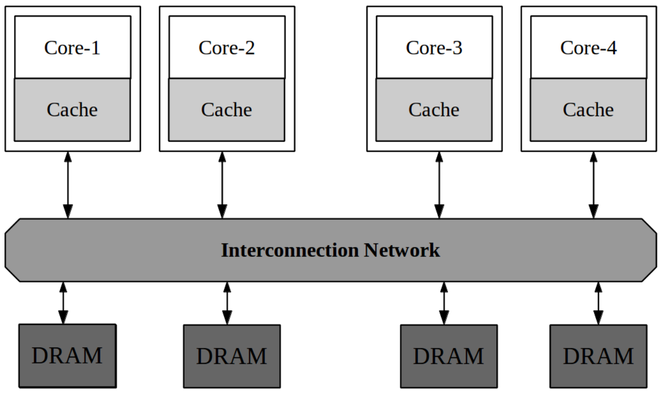


Figure 4.2: Multi-processors with distributed shared memory

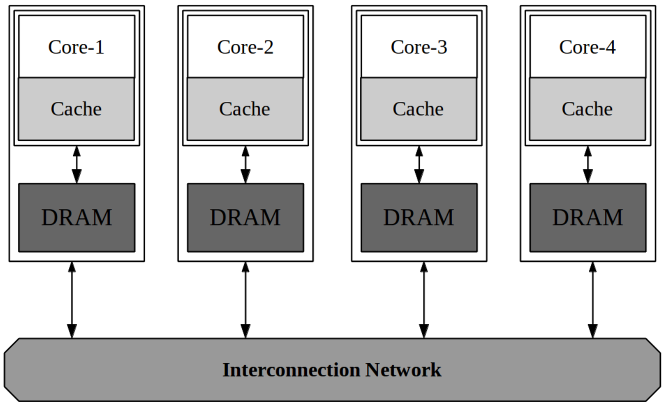


Figure 4.3: Distributed shared memory with message passing

CMP also called as multi-core processors and they are closely related to earlier multi-chip multiprocessor systems. The main difference between the multi-chip multiprocessor and chip multiprocessor lies in the communication network. Communication between two nodes in a multi-chip multiprocessor system can take hundreds of cycles because of messages travel through an inter-chip network as shown in Figure 4.1, 4.2, and 4.3. Generally off chip operations are clocked at a fraction of the chip frequency and are limited by on chip pin bandwidth. However in a CMP as shown in Figure 4.4, the communication messages between processor cores travel through an on chip interconnection network, capable of delivering much higher bandwidths at lower latencies. This significantly lowers the cost of inter node communication as compared to multi-chip multiprocessors.

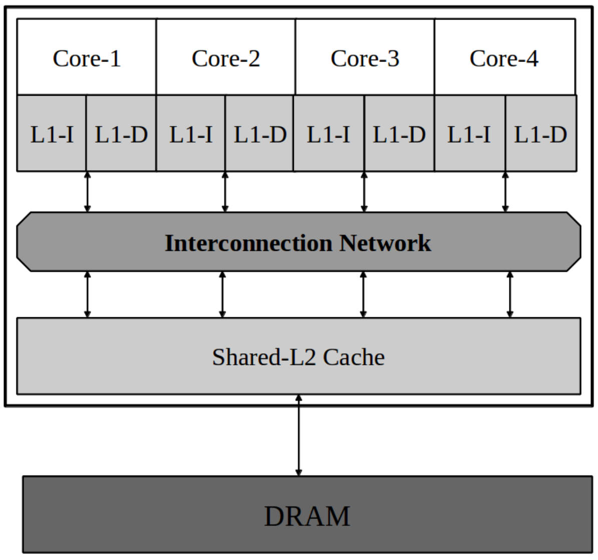


Figure 4.4: Chip Multiprocessor with on-chip shared L2 cache.

For any CMP, the memory system is a main component which can improve or reduce performance dramatically. The latest versions of many architectures are CMP with the last level of on-chip cache memory organized as either shared or private. Private L2 caches for each core has the advantage of low access latency, but these caches fail to make optimum use of on-chip memory space because some blocks may need to be replicated in other private L2 caches. While multiple cores with single shared caches make optimum use of on-chip cache space, they do suffer from high access latency compared to private caches. L Hsu and Iyer [110] have shown that organizing the last level L2 cache as a shared cache gives better performance than private caches. CMPs with last level caches as shared caches give rise to another type of miss that were not present in private caches: “inter-processor misses”. A miss is called an inter-processor miss in a multi-core system where a core evicts a block which was brought into the cache by another core and subsequent accesses by this core to the same block lead to a cache miss. To eliminate inter-processor misses, researchers have proposed many techniques: Shekhar [109] gives replacement ownership of a set to a core that brings the first block into that set. Only this core is allowed to evict the blocks from that set. In a multi-core system, ownership exists only for replacement; non-owner cores can read and write into the set but can’t evict the blocks. A major problem faced by such multi-core architectures is cache contention, where multiple cores compete for usage of the single shared L2 cache. Research shows that uncontrolled sharing leads to scenarios where one core evicts useful L2 cache data which belongs to another core.

This chapter proposes a fine grained control over the replacement ownership. Our work analyzes and proposes a technique to provide ownership of individual blocks in a set instead of providing ownership of a complete set to a core and it will be shown that such a fine control results in better utilization of the blocks inside a set. In the beginning, we investigated and presented a comparative understanding of cache misses in the context of CMPs with shared last level cache by comparing the CII scheme (Compulsory, Inter-processor and Intra-processor) to the traditional 3C scheme proposed for uniprocessors. This classification provides an insight into the interaction between cache references made by different cores

Then, we presented two different approaches for dealing with data ownership in the shared L2 cache and make the following important contributions in this Chapter:

- First, we proposed a novel technique called block pinning which associates cache blocks with owner processors (ownership in this chapter refers to right of a processor to evict blocks within the set on a cache miss) and redirects blocks that would lead to inter-processor misses to a small Processor Owned Private (POP) cache. Each core has its own POP cache. Also provided is a quantitative analysis of the effect of block pinning on both inter-processor misses and intra-processor misses in a shared cache.
- Then, as an enhancement over the set pinning and block pinning approach, we proposed a technique called adaptive block pinning which improves the benefits obtained by set and block pinning, by adaptively relinquishing ownership of pinned blocks within sets. The adaptive block pinning approach mitigates the effect of dominated ownership of blocks within a set by a few processors which is observed in the block pinning approach.
- Finally, we have evaluated each of the above approaches using a full system simulator which provides a characterization of the sensitivity of performance to various configuration parameters. In addition, we compare our approach to a set pinned cache [109] and a traditional cache.

The rest of the Chapter is structured as follows. Section 4.2, lays out the motivation for this work by analyzing the problem of inter-processor misses. In section 4.3, we describe the baseline architecture followed by basic taxonomy used in chip multiprocessors. Section 4.4 provides detailed explanation of the proposed architecture and ownership relinquishing techniques. Section 4.5 provides details of the experimental methodology used and also the details of the benchmarks applications used for evaluation. Results are presented in section 4.6 followed by related work in Section 4.7 and conclusion are given in section 4.8.

4.2 Motivation

4.2.1 Cache Miss Classification

One fundamental aspect of multi-core processors is the way in which the memory is organized. Memory architecture and its performance, influences both the performance of the tasks running on the processors as well the communication between tasks and processors. Especially when task's performance depends on the locality of data in caches. A smart memory miss classification along with its prevention can have a profound impact on performance and is yet to be explored for its efficiency in multi-core architectures.

4.2.2 Traditional Processors

The standard traditional cache miss classification with respect to single processor architectures is the 3C miss classification: Compulsory, Capacity and Conflict misses. According to this classification, cache misses are broadly divided into compulsory and non-compulsory misses. Compulsory misses are those misses that are generated due to initial reference to a memory location. The variation in the size of the cache as well as in the associativity makes negligible variation in the number of compulsory misses. Prefetching can help here, as can larger cache block sizes (which are a form of prefetching). Non-compulsory misses are classified as Capacity and Conflict misses where Capacity misses are those misses that occur regardless of associativity or block size. Capacity misses occur solely due to the finite size of the cache memory. Conflict misses are those misses that arises due to inadequate associativity (i.e., they do not occur in a fully associative cache). They usually have subclasses of conflict misses which are further categorized as mapping misses that are not avoidable given a particular degree of associativity as well as replacement miss that happen to be caused by a sub–optimal replacement policy. These classifications have enabled researchers to analyze the reasons for various classes of cache misses accurately. They have in turn influenced the successful development of a number of performance optimizations which target reduction of specific kinds of cache misses and improve system performance in the case of uniprocessors [8] [22][35][36][38].

4.2.3 Chip-Multiprocessors

The most current versions of several processor architectures include chip multiprocessors (CMPs) along with a shared L2/L3 cache [110] [18] [10]. In these CMPs, the processors compete for the shared cache. With regards to CMPs with shared caches, the traditional 3C miss classification is not really enough to comprehend and analyze the exact cause of cache misses. Traditional classification failed to model the contention that arises among different processors in gaining access to the shared cache. The opportunity to systematically characterize solutions to scale down misses in shared caches by making use of the existing classifications is also limited. While coherence misses are being utilized to model misses in multiprocessors with private caches, it aims to solve the problems associated with sharing data. To address these issues, a fresh cache miss analysis is required that interprets the interactions among transactions from several cores within a CMP along with shared cache. This is also crucial in order to develop various techniques to have highly effective shared cache management. So, we analyzed the identification of cache misses in the context of CMPs utilizing shared cache. Researchers are extensively working on managing shared caches in Chip Multi-Processors (CMPs). Different cache management schemes have been proposed for multi-core shared cache architectures. M. Dubois [110] first introduced a class of misses that was not present in the traditional processors. This category is called coherency misses and is present only in Multi Processors. These misses occur because of invalidation of cache blocks shared between private caches of multiple processors. Shekhar [109] introduced another way of categorizing misses in multi-core systems namely into compulsory misses, intra-processor misses and inter-processor misses (CII). The inspiration for our work comes from the transactions indicated in Table 4.1 and 4.2. Think about a CMP with two cores, Core1 and Core2 long with a fully associative shared L2 cache. Table 4.1 and 4.2 illustrate two possible forms of transactions which could cause a miss within the shared cache. Table 4.1 symbolizes a conventional capacity miss in which the same Core1 is accountable to each of the initial reference as well as expulsion of the stored memory block A.

Table 4.1: Miss due to eviction of a block by the same core

Intra Processor Misses		
Cores	**Access**	**Action**
Core1	Access location A	Cold Miss, location A brought to cache
Core1	Access location A	Cache hit, no change
Core2	Access location A	Cache hit, no change
Core1	Access location B	Replace location A, B brought to cache
Core1	Access location A	Cache miss

Table 4.2 also depicts a miss by processor Core1 that occurs to a memory element A. The difference here is that A was brought into the cache by an earlier reference by processor

Core1, but it was evicted by Core2, because of a reference to a different memory element B that is mapped to the same cache block as A.

Table 4.2: Miss due to eviction of a block by the different core

Inter Processor Misses		
Cores	**Access**	**Action**
Core1	Access location A	Cold Miss, location A brought to cache
Core1	Access location A	Cache hit, no change
Core2	Access location A	Cache hit, no change
Core2	Access location B	Replace location A, B brought to cache
Core1	Access location A	Cache miss

In this example, simply by classifying these kinds of misses as "capacity misses" just like in the 3C miss classification, we were unable to learn about the inherent dissimilarities within the cause for these types of misses. This is also true with conflict misses. Hence, an appropriate classification of the cache misses identical to that illustrated in table 4.1 is known to be Intra-processor misses as well as other one identical to that illustrated in table 4.2 is known to be Inter-processor misses. Therefore, the cache misses within a multi-core processor along with a shared cache are classified into compulsory misses, intra-processor misses as well as inter-processor misses.

In an attempt to provide a much more comprehensive knowledge of the CII classification, we present the life span of a memory element as indicated within the state diagram in Figure 4.5. This state diagram could be described as life span of a memory element in CMP when using the shared cache during the execution of a computer program, presuming this program is running on a dual core processor. The similar diagram can be easily outstretched to any range of cores. As shown in state diagram, the memory element under observation is at first usually not referred by any core, therefore we consider to be among the Never Referenced state. At this instant, |the initial access by P1 or perhaps even core P2 will result in a compulsory miss and the state of memory element is changed from Never referenced to the Referenced for the first time within the life span of the element.

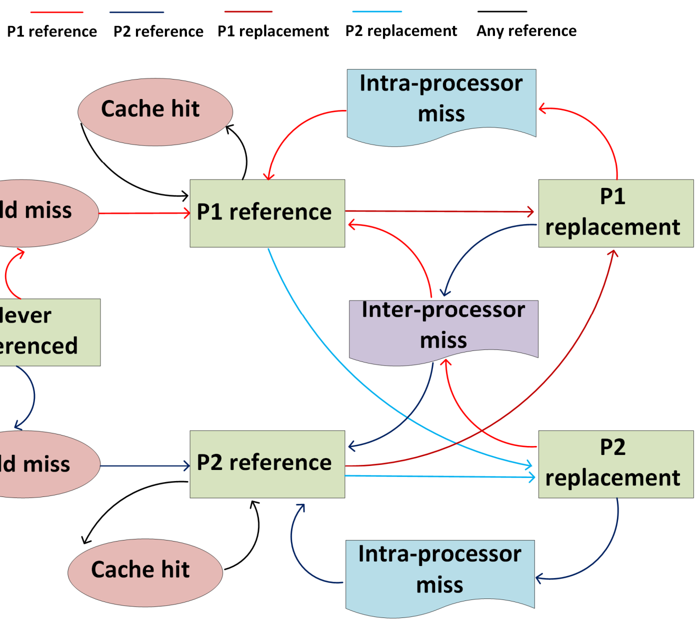


Figure 4.5: State diagram representing a memory element's life cycle in the shared cache

Now, further references by any of the core to this memory element in the Referenced state results to a cache hit. In case of replacement of the memory element, the state of the memory element changes into the Replaced state. And the memory location is marked with the core ID that have replaced the initial memory element. For illustration, a memory which is expelled from the cache due to a reference from core P1 is present in Replaced P1 state. At this moment, it is easy to recognize that each and every non-compulsory cache misses to any memory element take place while it is within the Replaced state. Therefore, the identification of the non-compulsory misses is based upon the fact that whether the cache miss is happening due to memory element actually being replaced by the same core P1 or possibly by a different core P2. This is inferred merely by matching the core suffering from the miss with the ID of the memory element in the Replaced state. It is important to understand that the identification of non-compulsory misses into intra-processor misses as well as inter-processor misses is orthogonal with the identification of the same as capacity and conflict misses. As an illustration, the examples presented with reference to table 4.1 and table 4.2, in case of a fully associative cache, represent (a) capacity miss that is also an intra-processor miss and (b) capacity miss that is also an inter-processor miss. Conflict misses may also be classified as intra-processor misses and inter-processor misses by this classification. This CII classification is a bit more significant in comparison with the 3C miss classification and more importantly, it is able to model the correspondence within transactions of several cores at the level of the shared cache.

4.2.4 Characterization of Compulsory Inter-processor and Intra-processor misses

We have measured the distribution of various types of misses. Figure 4.6 plots the distribution of compulsory, inter-processor and intra-processor misses with our baseline system configuration (A detailed description of our baseline configuration is given in Section 4.4). The black portion of the stacked bars represents the inter-processor misses, the spotted portion (in the middle) represents intra-processor misses and the striped portion represents the compulsory misses. On an average, 40.3% of the misses are inter-processor misses, 24.6% of the misses are intra-processor misses and the remaining 35.1% are compulsory misses.

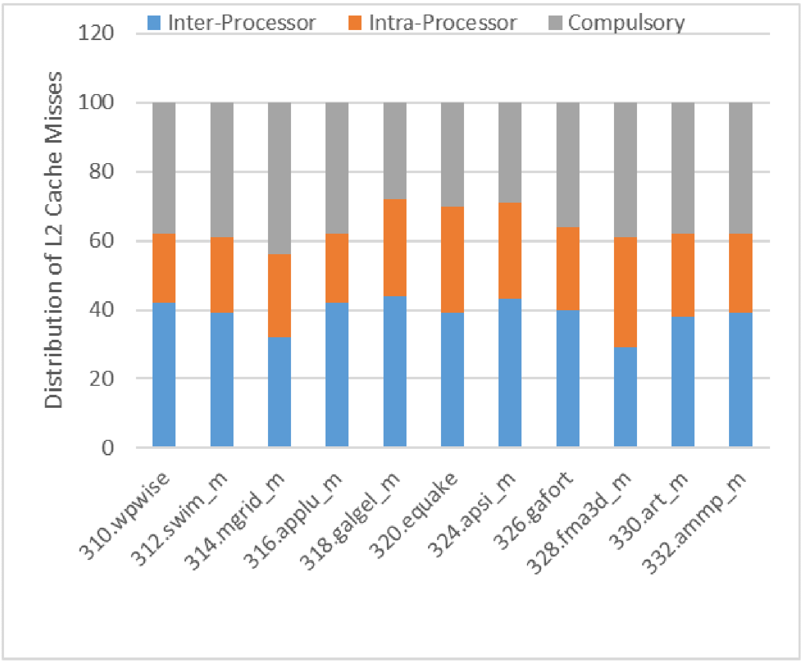


Figure 4.6: Distribution of compulsory, inter-processor and intra-processor misses [109]

Now, reducing off–chip accesses is the key to a successful shared cache management scheme in a CMP with large shared L2/L3 cache [19]. The effect of compulsory misses can be reduced by hiding their latency. This can be achieved by prefetching data into the cache before it is accessed. There have been many recent studies for reducing memory bandwidth and the number of off–chip accesses through hardware/software data prefetching [27] [48]. However, the focus of this chapter is on developing techniques to reduce inter-processor and intra-processor misses. In our proposed architecture, inter-processor misses are eliminated by giving replacement ownership of a block to a processor, while Shekhar [109] eliminates inter-processor misses by giving replacement ownership of a set to a processor.

For a "hot set" [109] in the on-chip cache, ownership of the complete set is given to a single processor. But if a set is not a "hot set", providing ownership to a single processor will increase the load on POP caches of other processors. Figure 4.7 indicates that only about 9% of the memory addresses result in hot sets, so the number of hot sets is not going to be too large.

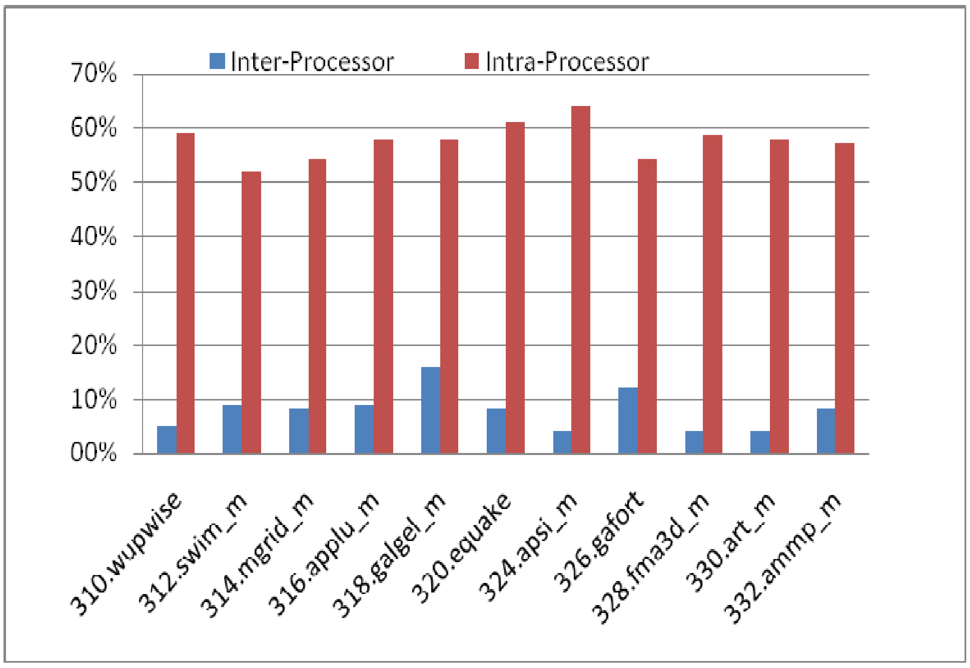


Figure 4.7: Memory addresses leading to Inter and intra-processor misses [109]

4.3 Taxonomy Used in CMPs

The most common cache miss classification scheme for single processor architectures is the 3C miss classification: Compulsory, Capacity and Conflict misses. It can be broadly classified as compulsory and non-compulsory misses (conflict and capacity misses).

Compulsory Misses: Compulsory misses are those misses caused by the first reference to a datum. Cache size and associativity make no difference to the number of compulsory misses.

Non -Compulsory Misses:

Capacity Misses: Capacity misses are those misses that occur regardless of associativity or block size, solely due to the finite size of the cache.

Conflict Misses: Conflict misses are those misses that occur due to insufficient associativity (i.e., they do not occur in a fully associative cache).

Intra-processor Miss: Non-compulsory misses are further classified based on the processor responsible for evicting the referenced block. A non-compulsory miss is classified as an intra-processor miss if it was evicted by the same processor that brought it into the cache.

Inter-processor Miss: A non-compulsory miss is classified as an inter-processor miss if the block brought into the cache by one processor is evicted by other processors on the chip.

Hot Blocks: If the number of intervening references between successive references to few blocks in the L2 cache is large, then it indicates that these few blocks are accessed over and over again and we call these blocks as hot blocks**.**

Processor Owned Private (POP) cache: A very small region of the shared L2 cache, which is confined to be written by individual processors.

4.4 Baseline Architecture

The block diagram of the proposed block pinning architecture for L2 cache is shown in Figure 4.8. As seen from the figure, we have eight cores C_1 to C_8 on the same chip with individual private L1 caches and a large shared L2 cache. The L2 cache is further partitioned into a large shared cache and eight small POP caches (one for each core). In case of a hit, the common shared L2 cache behavior is similar to a traditional shared cache. In case of a miss in the common L2 cache, all the POP caches are searched in parallel. If there is a hit in any of the POP caches, the data block is transferred to the requesting core.

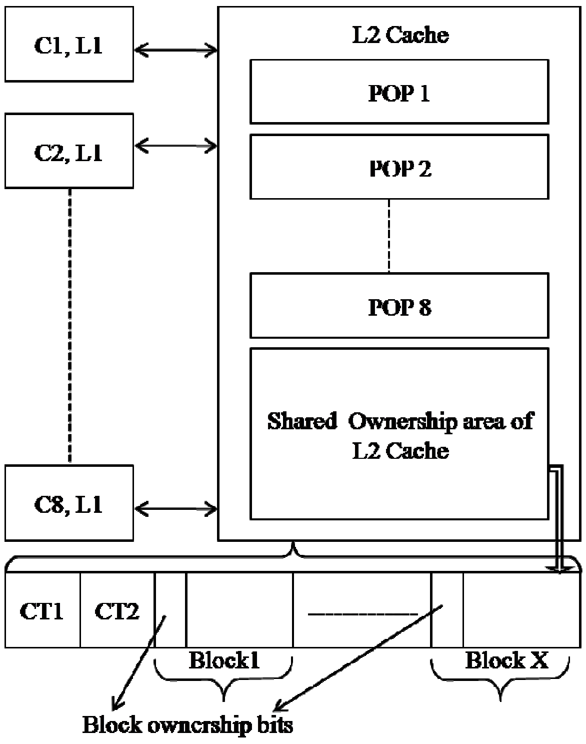


Figure 4.8: Block Diagram of Proposed Architecture

4.5 Shared Cache Management Scheme

This section, first presents the data ownership policy and some of its drawbacks that restricts it to achieve performance benefits at low implementation cost. Then, the rest of the section describes in detail the proposed block ownership management scheme for shared L2 cache.

4.5.1 Set Pinning Ownership Scheme

Set pinning is basically a cache management scheme in which each single core obtains replacement ownership associated with a certain number of sets within the shared L2 cache. Exclusively the processing core which has the replacement ownership of a given set actually being accessed have the permissions to carry out change in that set. This novel shared cache management scheme eliminates inter-processor misses without paying for additional costs of maximizing the associativity of a given shared cache. The conceptual proposal of the set pinning scheme is founded on two significant observations regarding the behaviour of non–compulsory misses within the shared cache. Researchers examined the total number of diverse memory addresses within the references that results in inter-processor as well as intra-processor misses.

The division of the number of references to diverse memory addresses leading to inter-processor as well as intra-processor misses have been measured and it has been observed that the low fraction of distinct memory addresses leading to inter-processor misses suggests that the majority of the inter-processor misses take place mainly because of few blocks within the memory. We also examined the amount of time period in relation to the total number of intervening references between successive references to each of these blocks and certainly noticed that the majority of blocks are accessed again and again within 100 references 64.5% of the time on average. This indicates that these blocks are frequently accessed and this increases miss rate. We have also observed that the policy of allocating ownership of sets to processors may lead to many blocks in the set being unused. Secondly, the policy of allocating sets to processors is based on first come first serve allocation. This simple allocation policy results in an unfair division of the sets in the shared L2 cache. So we have proposed a new cache management scheme to exploit these two observations. First, by disallowing the large number of references for these few blocks that are responsible for evicting L2 cache blocks and therefore causing inter-processor misses. Secondly, the issue of fairness in acquiring ownership in the shared L2 cache.

4.5.2 Proposed Block Pinning Scheme

Block pinning is a cache management scheme where every processor acquires replacement ownership of a certain number of blocks in the shared cache. Only the processor that has replacement ownership of the block being accessed can replace that block entry in the set. The basic flow chart explaining the logic of block pinning is shown in Figure.4.9.

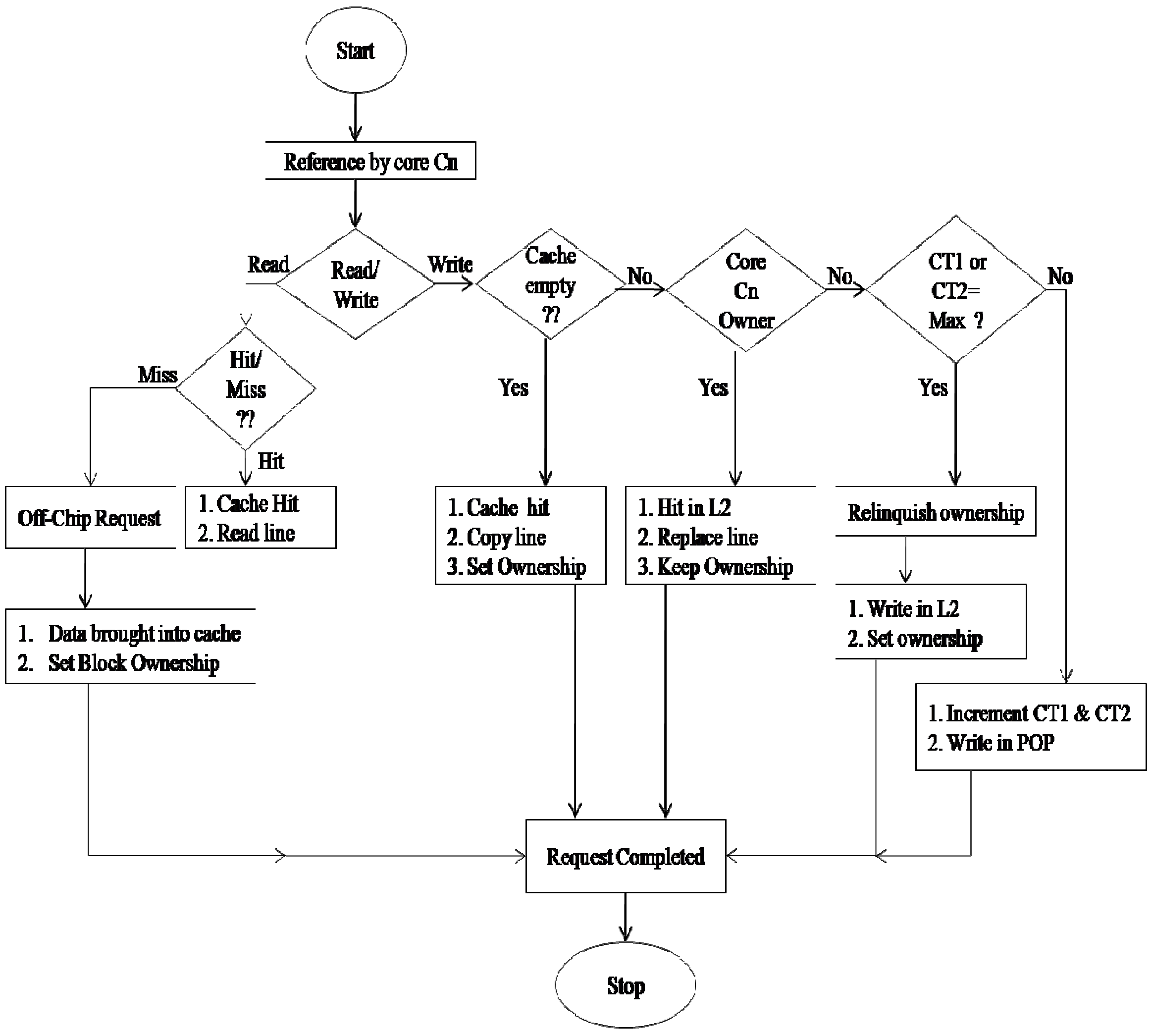


Figure 4.9: Basic flow chart explaining the logic of adaptive block pinning

In multi-core systems, inter-processor misses occur when a block (A) brought into the cache by one core (C_1) is evicted by another core (C_2) and any subsequent access by core (C_1) to the same block (A) leads to a cache miss. A simple method to prevent inter-processor misses is by allocating block ownership to a core at the time of bringing data into the cache from off-chip memory. This method assigns replacement ownership to all the blocks in the entire L2 cache. While all cores can read and write into the block, only the owner core has the permission to replace or evict a block from the cache. For example, assume a dual core processor with cores (C_1) and (C_2) as shown in Figure. 4.10, where both the cores are sending references to the same set. In the absence of block ownership, if C_2 experiences a miss, it may evict a block which was brought into the cache by C_1. Now, any subsequent access by C_1 to the same block will result in a cache miss and lead to an overall increase in the miss rate. But if block replacement ownership is assigned to cores, C_2 will not be able to evict a block that is owned by C_1 as shown in Figure 4.10. One particular observation with this method of reducing inter-processor misses is that it may lead to an increase in intra-processor misses.

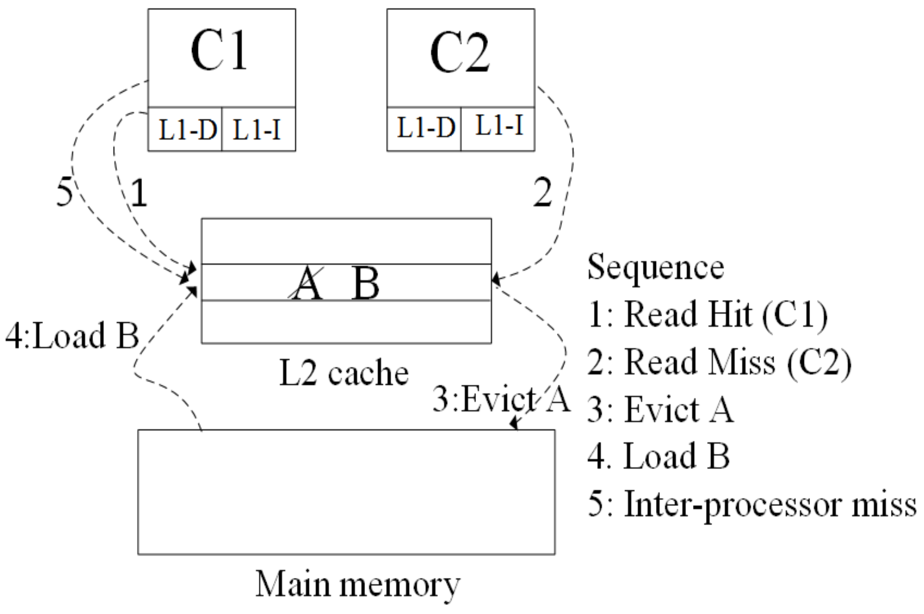


Figure 4.10: Inter processor Miss in dual core processor

The rate of intra-processor misses can be controlled by identifying whether the referenced set is a “hot set” or not. Once the set is identified as hot, assigning new block ownership in the hot set will increase intra-processor misses, since each core has lesser number of replacement candidates to choose from when it requires more blocks in that set. So, hot set miss rate is high either due to inter-processor misses or due to increased intra-processor misses.

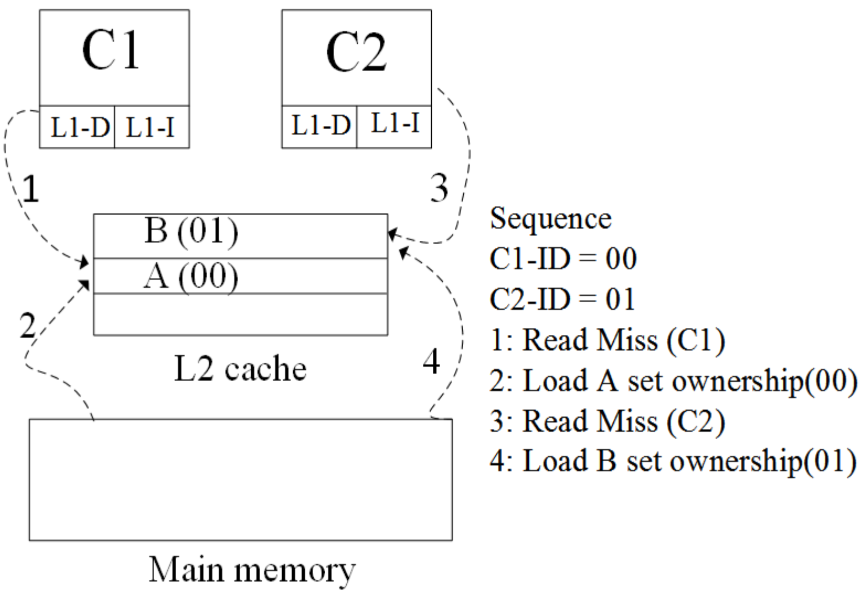


Figure 4.11: Allocation of block ownership to prevent eviction in dual core processor

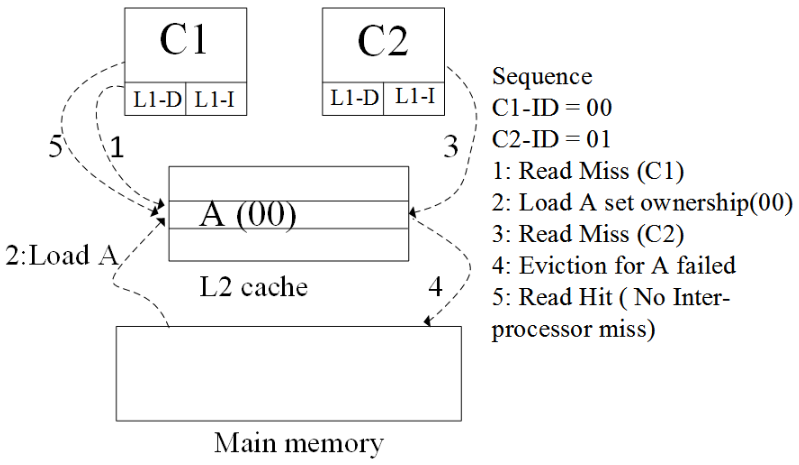


Figure 4.12: Ownership prevent eviction in dual core processor

To control (reduce) this increase in intra-processor misses, POP caches are used. Suppose, if during the last N_1 accesses to a particular set there are M_1 or more misses (where M_1 is the threshold value), then this particular set is considered to be a “hot set” and the ownership of one of the cores is canceled and the core will now bring its blocks from memory to its POP cache instead of the hot set. This process will decrease the traffic to the hot set and eventually the miss rate will come down. This process of canceling the ownership of cores from a particular set may lead to a situation where only one core owns all the blocks in a set. To avoid this situation, the ownership of a core is canceled if it owns a certain minimum number of blocks. If a set is not a hot set, it means that not many addresses are being generated by different cores that index into this set. In this case, to reduce the miss rate, the proper distribution of block ownership among the cores is necessary. Consider an example when core C_1 owns most of the blocks in the cache and is rarely using these blocks while core C_2 has ownership of a few blocks and suffers misses in that set because it has fewer blocks to choose from when evicting a block. If the ownership of the less frequently used blocks of core C_1 is transferred to core C_2, then the overall miss rate can be controlled and hence reduced. The algorithm applied for relinquishing the ownership of blocks is explained in Algorithm-1 and implemented using full system simulator. Also by allowing all the cores in a multi-core system to share "non hot sets", the load/traffic on the POP cache can be reduced. Now, to assign block ownership in the last level shared cache, ($\log_2 n$) bits in each block are needed to indicate owner of the block, (where n is number of cores in the multi-core system). When for the first time, a core fetches a block from off-chip memory to the cache, its '*Core ID*' number will be written in the ownership bits of the block. Now only this core has the right to evict the block from the cache, as long as keeps ownership of the block.

4.5.3 Cache HIT/MISS Policy

In the proposed cache architecture, the shared L2 cache is organized as POP caches and a common cache. In case of a miss in the L1 cache, the request is forwarded to the common L2 cache. If there is a hit, then the requested block is sent to the requesting core. In case of a miss, the POP caches of all the cores are probed in parallel for the requested block. If the request hits in one of the POP caches, then the block is sent from that POP cache. These two partitions are non-inclusive in nature. Whenever a cache miss occurs in the shared Last level cache, it may result due to any one of the following scenarios:

1 The reference from a core may point to a set where some of the blocks are not owned by any of the cores in the multi-core system. In this case, the requested block will be transferred from memory to the referenced set and ownership bits will be set with the '*CoreID'* of the requesting processor.

2 The request from a core to a block address may point to a set where all the blocks are owned by cores other than the one which experiences a miss. In this case, a block cannot be replaced from this set because the requesting core doesn't own any block. So, data from memory will be transferred to the POP cache of the requesting core that is experiencing a miss.

3 The reference from a core to a lock address is pointing to a set where the requesting core owns some of the blocks in that set. In this case, the core will replace one of the blocks

owned by it with the new block. In this case, the block to be replaced is one which is least recently used among the blocks owned by the core in that set, which need not be the least recently used block of the entire set.

4.5.4 Block Ownership Relinquishment Policy

This section proposes two methods to relinquish the ownership of a block: In the first method, one saturating counter per block is used. This counter is initialized to half of the maximum count. Every time when the block is accessed and it results in a hit, the counter value is increased by one. If the counter reaches maximum value i.e. all 1's it will stay there (saturating). If a processor experiences a miss in a particular set, then the counters corresponding to all the blocks owned by other processors in that set are decremented by one. If any counter hits zero, ownership of this block is cancelled and given to the processor whose miss makes the counter hit zero. Qualitatively, a counter hitting zero means that the processor owning it is not using it effectively and this block can be used more effectively by other processors. This technique has a major drawback in that the numbers of counters required is equal to the number of blocks in the cache. This huge hardware requirement makes this technique less attractive.

The other technique for ownership relinquishment requires just two counters per set (CT_1 and CT_2). The algorithm for this technique is given below. CT_2 is used to determine whether or not a set is a "hot set" and CT_1 is used to fine tune the number of blocks owned by each processor in a set. Selecting two counters is based on the observation that misses may occur in a set can rise because of two reasons:

1. Set is a "hot set" and most of the processors are trying to put their blocks in the same set and hence intra-processor misses are more.
2. Set is not a hot set but the distribution of blocks in the set is unfair, i.e. the processor requiring more blocks owns less blocks and the processor owning more blocks is not utilizing them effectively.

Let the set Cores = $\{C_0, C_1,..C_7\}$ represent the cores present in the baseline system. The set HS includes all the "hot sets" of the cache. The ownership of the different blocks in the set is indicated in the Owners set. $Owners^{(s)}$ denotes the set containing the owners of all blocks in set s. x^j, y^j, m^j, n^j are chosen by experiments to meet the performance needs of Application j. The shorthand numAccesses(s) is used to denote the number of accesses to set s. The set RB contains the list of all blocks that have a particular core, say core c as their owner.

/* Algorithm for relinquishing ownership and cache operation */

Algorithm 1: Algorithm for relinquishing ownership

```
function handlePinnedCacheMiss
INPUT: Requesting core (c), Referenced Set (s), HS, Owners$^{(s)}$, $CT_1^{(s)}$, $CT_2^{(s)}$.
OUTPUT: $HS_{new}$, $Owners_{new}^{(s)}$
BEGIN
1.      $X \leftarrow x^j$, $Y \leftarrow y^j$, $M \leftarrow m^j$, $N \leftarrow n^j$, $CT_1^{(s)} \leftarrow 0$, $CT_2^{(s)} \leftarrow 0$;
2.      if (numAccesses(s) == Y && $CT_1^{(s)}$ == X) //unfair distribution
3.      $B_k^{(s)} \leftarrow$ LRUBlock(s);
4.      $B_k^{(s)}$. Owner $\leftarrow$ c, update Owners$^{(s)}$;
5.      $Owners_{new}^{(s)} \leftarrow Owners^{(s)}$, $CT_1^{(s)} \leftarrow 0$;
6.      endif
7.      elsif (numAccesses(s) == Y && $CT_1^{(s)}$ != X) // relinquishment not needed
8.      $CT_1^{(s)} \leftarrow 0$;   // reset counter
9.      endif
10.     elseif (numAccesses(s) == N && $CT_2^{(s)}$ == M)  // set is ‘hot’
11.     HS.add(s), $HS_{new} \leftarrow$ HS;
12.      while (missRate >= MRT) || (s.numBlocksWithOwner (c) != numBlockInSet(s)))
13.     for some k   Cores and k != c
14.     RB $\leftarrow$ findBlocksWithOwner(k);
15.     $\forall$ r  $\in$  RB,  r.Owner $\leftarrow$ xx; // Cancel ownership
16.     update Owners$^{(s)}$, $Owners_{new}^{(s)} \leftarrow Owners^{(s)}$,
17.     k.loadNewBlockLocation $\leftarrow$ $POPCache_k$
18.     end for
19.     end while
20.     end if
21.     elseif (numAccesses(s) == N && $CT_2^{(s)}$ != M)
22.     $CT_2^{(s)} \leftarrow 0$;
23.     end if
END
*********
*********
```

Algorithm 2: Algorithm for cache operation

```
function handleReference
INPUT: Read/Write request ($Req_j$) from some c   Cores that indexes set s.
BEGIN
Lookup $L1_c$
  if (hit)
Read/write data block, update LRU stack
  else          // L1 miss
        $FwdReq_j \rightarrow BlockPinnedL2_c$
        if (hit)
                Read/write data block, update LRU stack
        else   // Pinned L2 cache miss
```

```
                $CT_1^{(s)}$++, $CT_2^{(s)}$++;        //increment miss counters
                handlePinnedCacheMiss;    // Algorithm 1
                FwdReq $_j$ →POPCache$_c$ ;
                Lookup POPCache$_c$ :
                if (hit)                // POP Cache hit
                    Read/Write data
               else
                   FwdReq $_j$ →off-chip ;
              end if
        numAccesses(s)++;
        end if
    end if
END
*********
*********
```

These blocks will be relinquished to bring the cache miss rate below a predetermined Miss Rate Threshold (MRT). $B_k^{(s)}$ is the LRU block located in the kth way of the set s. Counter CT_1 produces a high output (all 1's) if there are X misses in the last Y accesses to a set and counter CT_2 produces a high output (all 1's) if there are M misses in the last N accesses to that set. Here M is a multiple of X and N is a multiple of Y. Multiplication factor in both cases is the same. So, if the miss rate increases above a particular value, CT_1 will detect it first, and the set is assumed not to be a "hot set" at this point. The ownership of the blocks in the set which are not being utilized effectively is canceled. To do this, whenever CT_1 produces a high output as shown in Figure. 4.13 ownership of the least recently used block in the set is cancelled, so that a processor suffering more misses can acquire the ownership of this block and the miss rate can come down.

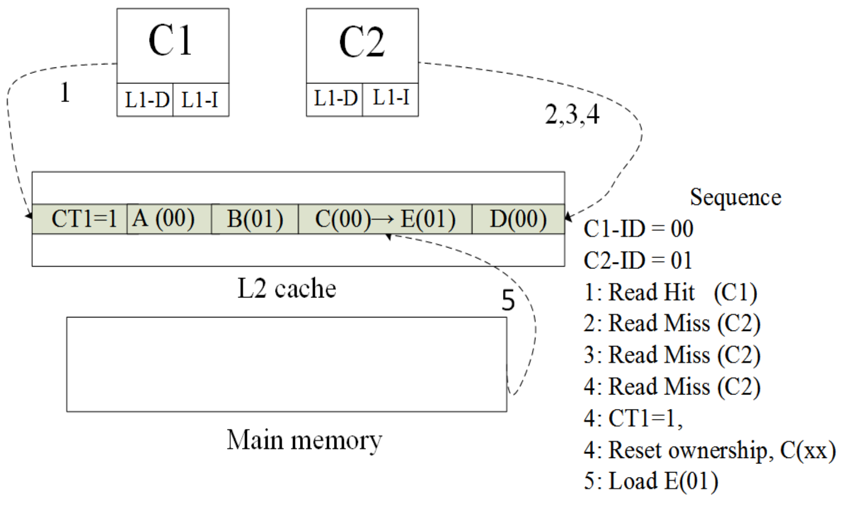


Figure 4.13: Cancellation of block ownership in dual core processor

Qualitatively, in canceling ownership of the least recently used block, it is assumed that this block is not being utilized properly by the owner and can be better utilized by processors other than the current owner. Once ownership of a block is canceled, CT_1 is reset to its initial value. If the miss rate still remains high after a few such attempts, the number of such attempts as determined by the ratio of N/Y, CT_2 will also produce a high output and the set is treated as a “hot set”. This indicates that every processor is trying to put its blocks in this set. In this case, the ownership of all the blocks of a particular processor is canceled as shown in Figure. 4.14 and this processor will now bring any new blocks to its POP cache instead of the “hot set”. This cancelation of ownership of blocks will continue until either miss rate goes below the threshold value or the complete set is owned by a single processor. In effect, the load on the hot set and the miss rate both will reduce.

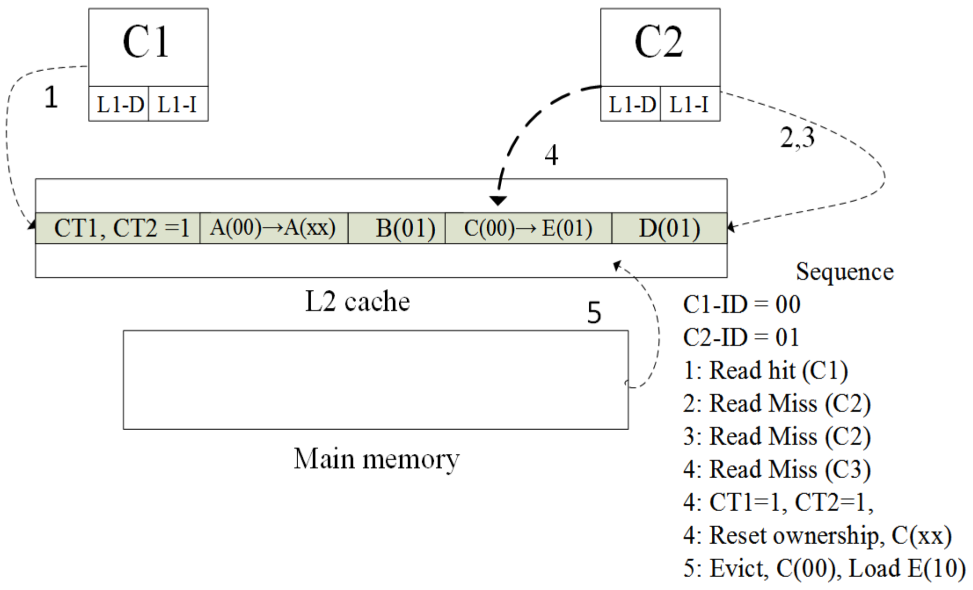


Figure 4.14: Counter CT-1, 2 saturates and ownership bits of C1 are reset in dual core processor

4.5.5 Hardware Support

The relinquishing of the blocks in a set by an owner core is based on the confidence counters for each set (CT_1 and CT_2), which indicates the confidence of the system in assigning ownership of a block to the current owner. The total additional hardware cost includes that for the counters CT_1 and CT_2 along with the processor identifier field (*CoreID)* for the block pinning architecture. After experimenting with a range of values from 2 to 16 and we found that 4 bits for CT_1 and 6 bits for CT_2 were sufficient to account for the longest duration of ownership without frequent saturations. Therefore, the total additional hardware cost is about 2.5% of the L2 cache in our baseline configuration.

4.6 Experimental Methodology

In this section we describe our baseline system configuration and evaluation methodology. All the results are obtained with the baseline system configuration described below.

4.6.1 Simulation Environment

Evaluating the performance of CMPs with different Last level cache architectures requires a way of simulating the environment in which we would expect these architectures to be used in real systems. We have used Virtutech Simics [99] full system functional simulator with modified gcache extended with Multifacet GEMS [100]. The base line configuration is given below in Table 4.3.

Table 4.3: Configuration Parameters for simulation

No of Cores	8
Core Mode	Single Thread
Frequency	1Ghz
L1-Data Cache	32kb, 64 bytes
L1-Instruction Cache	32kb, 64 bytes
Shared L2 Cache	8-Way Set Associative
L2- Cache (Size)	2MB
POP Cache	8-Way, 16Kb, 64bytes

4.6.2 Benchmarks

To quantitatively examine the CII classification and to figure out the benefits of the proposed block pinning as well as adaptive block pinning schemes for shared cache memories on CMPs, we put into use few programs from the SPEC benchmark suite [106]. All of the chosen programs make use of the reference input set and certainly fast forwarded to the beginning of the main loops. We have also used selected programs from the PARSEC [108] benchmark suite. All of these benchmarks use sim-large inputs and are fast forwarded to the beginning of Region of Interest (ROI). The method for the simulations involves first skipping both the initialization and thread creation phases and then fast-forwarding while warming up the cache for 500 million cycles and then collect statistics until the end of another 500 million cycles.

4.7 Results

This section analyses the impact of our novel adaptive block pinning technique for block ownership technique on performance in the baseline architecture. Figure 4.15 shows the performance improvement achieved with adaptive block pinning when compared to a conventional cache. On average we observed that there is a significant reduction in misses per thousand instructions (mpki) while running the PARSEC benchmark applications. As stated earlier adaptive block pinning does not influence the number of compulsory cache misses. Adaptive block pinning eliminates inter-processor misses but they may introduce few additional intra-processor misses in the POP cache. Therefore the effective misses are the misses that occur in both block pinned L2 cache as well as POP caches. The effective miss rate is defined as

$$EffectiveL2 - Missrate = \frac{BlockPinnedL2misses \cap POPCachemisses}{TotalL2accesses}$$

The effective miss rates for adaptive block pinning and set pinning, normalized with respect to the miss rates of the traditional shared cache scheme are plotted in Figure. 4.15.

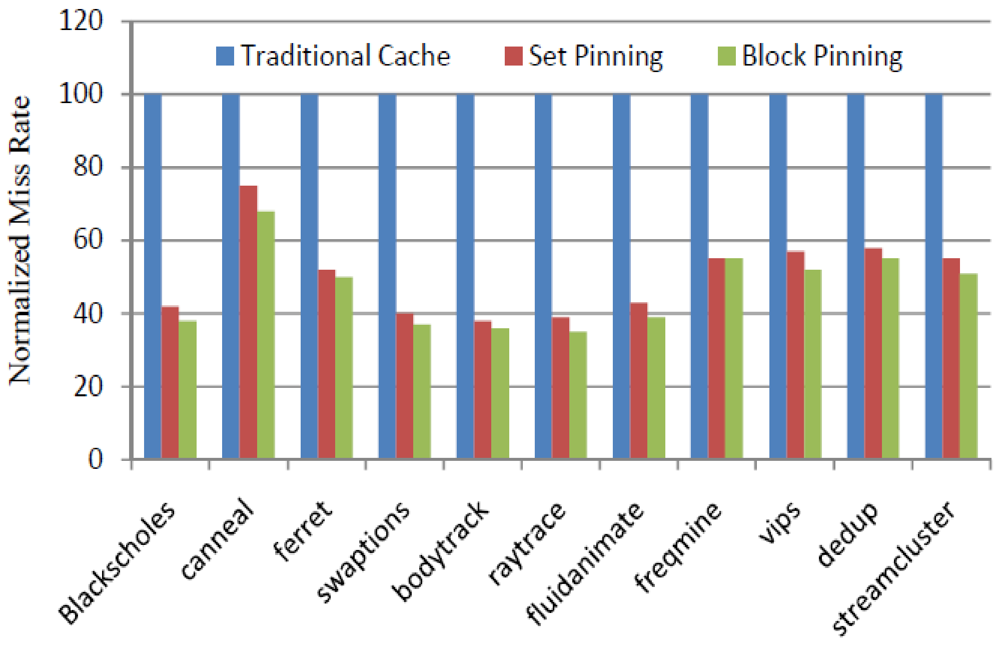


Figure 4.15: L2 Cache Miss Rate

The percentage of improvement is obtained by taking the difference between the average value along all the applications for reference and proposed schemes. The adaptive block pinning scheme achieves an average improvement of 22% and 4% as compared to traditional cache and set pinned cache schemes. Another metric that determines the performance of our scheme is the effective hit rate of the POP caches. We define this metrics as

$$HitRate - POP = \frac{TotalHitsinPOPCaches}{MissesinBlockPinnedL2}$$

The effective hit rates in POP caches are plotted in Figure 4.16.

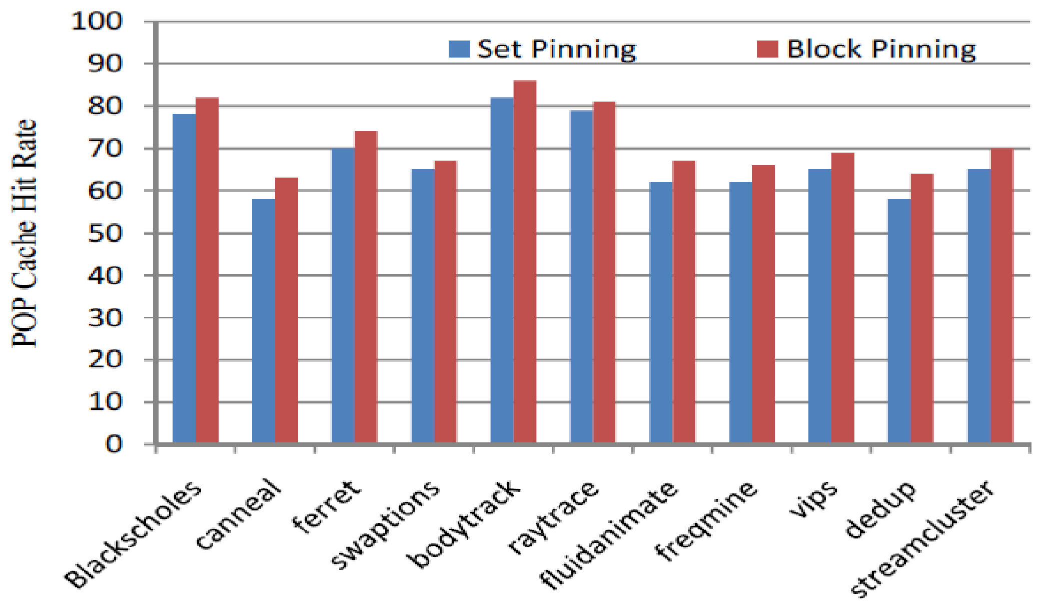


Figure 4.16: POP Cache Hit Rate

The hit rates are found to improve by 3-4% (averaged across all benchmarks) as compared to the set pinning scheme. The sensitivity analysis with varying number of cores plotted in Figure 4.17 shows the speedup obtained by our adaptive block pinning scheme relative to the traditional shared cache and set pinning scheme with 4, 8 and 12 cores.

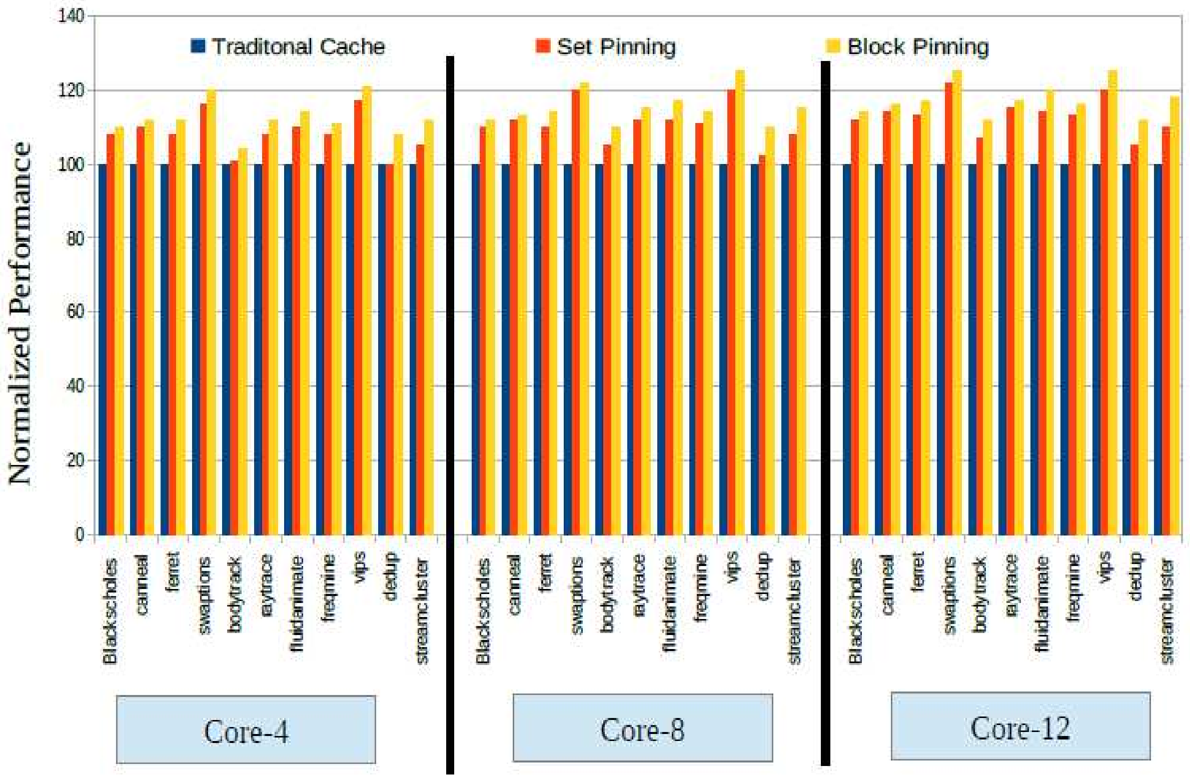


Figure 4.17: Performance with different cores

4.8 Related Work

Qureshi [54] divides the blocks in a set among different processors. Here, at the end of a time frame, miss rate is measured, which means that any action to reduce the growing miss rate can be taken only at the end of a time frame. This paper proposes an implementation where corrective action can be taken at any time when miss rate grows above a given threshold value. Recently, people from research and academia have investigated several multicore cache architectures in the effort to attain the reduced access latency of private L2 caches together with the reduced off-chip miss rates of shared L2 caches [60] [114] [116]. Dynamic and Static last level shared cache management policies have been investigated in an effort to take care of the problem of data isolation [54]. Researchers have also extensively analyzed quite similar issues in distributed video-on-demand systems. Victim replication [116] is basically a modification of the shared last level cache design that attempts to maintain copies of local primary cache victims inside the local L2 cache portion however permits a number of copies of a cache block to co-exist in various L2 portions of the shared L2 cache. Chang and Sohi [114] present CMP Cooperative Caching, a simple setup to control |total on-chip cache resources as well as incorporates the merits of both private as well as shared cache organizations by creating an aggregate “shared” cache, by means of cooperation in between private caches. The technique used by cooperative caching is to keep a locally expelled block within the on-chip L2 cache of a different private portion which may free cache space instead of evict it from the on-chip hierarchy entirely. With regard to set–pinning as well as adaptive set–pinning, Shekhar [109] diverts the cache blocks which may trigger an existing cache block to be evicted from the on chip cache towards the small POP cache to decrease off–chip accesses, and hence avoiding inter-processor misses as well as reducing intra-processor misses in CMPs. Adaptive Selective Replication (ASR), dynamically tracks the workload patterns in order to manage block replications in the cache and it was suggested by Beckmann and Wood [60]. The ASR mechanism replicates cache blocks in the event when it estimates that the benefits of replication in terms of much lower L2 hit latency exceed the expense as a consequence of elevated L2 misses. Adaptive Selective Replication may work extremely well in association with our scheme to further improve the L2 hit latency. Petoumenos [117] has implemented} a better statistical model of a CMP shared cache which explains each of the cache sharing as well as its management using a novel fine-grained technique called StatShare. This model precisely explains the behavior of shared threads using run-time statistics and enables us to learn how systematically each thread uses its space. Even though this model precisely identifies capacity misses and can approximate cold misses however it fails to address conflict misses. Software level shared cache management policies for CMP have been explored during the last few years. Rafique [118] presented an Operating System-driven which typically incorporates a hardware cache quota management technique, an OS interface as well as a set of OS level quota orchestration scheme to obtain enhanced flexibility. Tam [70] addressed the problem of uncontrolled sharing and presented a software assisted technique in the Operating System which enables splitting up of the Last level shared L2 cache by governing the assignment of physical pages. These software schemes provides higher flexibility at the expense of inhibited applicability as compared to a hardware scheme. In uniprocessors, the minimization of conflict misses in privately used caches connected with a single core continues to be a useful problem of investigation and there have been a variety of vital works that manage this challenge in both of the hardware and software [119] [48] [123] [124]. Collins and Tullsen [119] revealed the usage of a hardware miss classification table which permits the processor or memory controller to distinguish every individual cache miss as either a conflict miss or just a capacity (non-conflict) miss.

4.9 Summary

Inter-processor misses constitute 40% of the total number of misses in a Chip Multi Processor with shared L2 cache. This work proposes a new architecture to eliminate these misses without a significant increase in intra-processor misses by giving replacement ownership of a block to one of the processors. This work also shows that if a processor is not utilizing blocks owned by it optimally, the ownership of its blocks can be transferred to other processors. In this work, two techniques to relinquish the ownership of a block are presented. The first technique uses a saturating counter per block that is decremented whenever a request misses in the set. Ownership of a block is relinquished when the counter hits zero. Since the first technique incurs a significant hardware overhead, a second technique that uses two counters per set (CT_1 and CT_2) is proposed. CT_2 is used to determine whether or not a set is a “hot set” and CT_1 is used to fine tune the number of blocks owned by each processor in a set.

Chapter 5

Selective Replication in the Shared Last Level Cache

This chapter presents Selective Replication scheme in Shared Last Level Cache for effectively dealing with fixed block location problem in NUCA caches.

Chapter 5

Selective Replication in the Shared Last Level Cache

5.1 Introduction

In the previous chapter of this thesis, we have focused on the problem of inter-processor and intra-processor cache misses in the shared L2 cache for large-scale CMPs. For that study, we have assumed a shared L2 cache organization with a uniform access latency and physical mapping of blocks to uniform shared L2 cache. In this chapter, we discuss the perks and drawbacks of this organization, and we propose an alternative mapping policy. As discussed in the introduction of this thesis, an important decision when designing a multi-core processor is how to organize and manage the last-level on-chip cache, i.e., the L2 cache in this thesis, since cache misses at this cache level result in long-latency off-chip accesses. The two common ways of organizing this cache level are *private* to the local core or *shared* among all cores. Figure 5.1 presents the trade-off between two conflicting goals that is to reduce off chip miss rate and to reduce on chip miss latency.

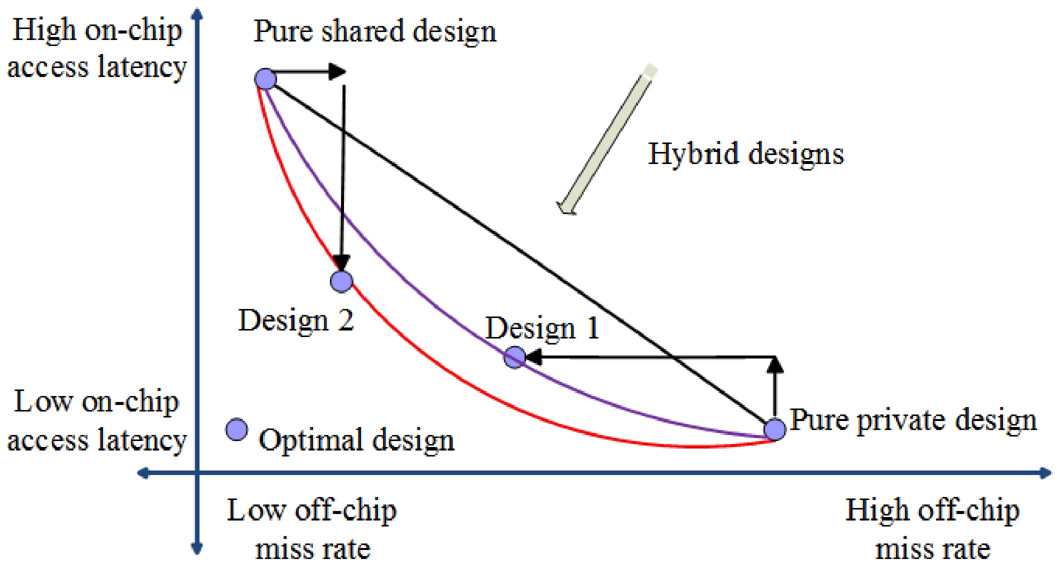


Figure 5.1: Trade-off between off-chip miss rate and on-chip access latency in private/shared on-chip cache designs

The private L2 cache organization, ensures fast access to the L2 cache. However, it has two main drawbacks that could lead to an inefficient use of the aggregate L2 cache capacity. First, local L2 banks keep a copy of the blocks requested by the corresponding core, potentially replicating blocks in multiple L2 cache banks. Second, load balancing problems appear when the working set accessed by all the cores is heterogeneous, i.e., some banks may be over-utilized while others are under-utilized. Since these drawbacks can result in more off-chip accesses, which are very expensive, there is a shift from private caches to shared cache organization. However, the non-uniform latencies in a single large shared cache becomes the bottle neck for this kind of architecture. Therefore, researchers from both industry and academia proposed to implement a shared non-uniform cache organization for multi-core processor [10] [76]. The shared L2 cache organization, also called non-uniform cache architecture (NUCA) [19] as shown in Figure 5.2, which provides more efficient use of the L2 cache by storing only one copy of each block and by distributing the copies across the different banks.

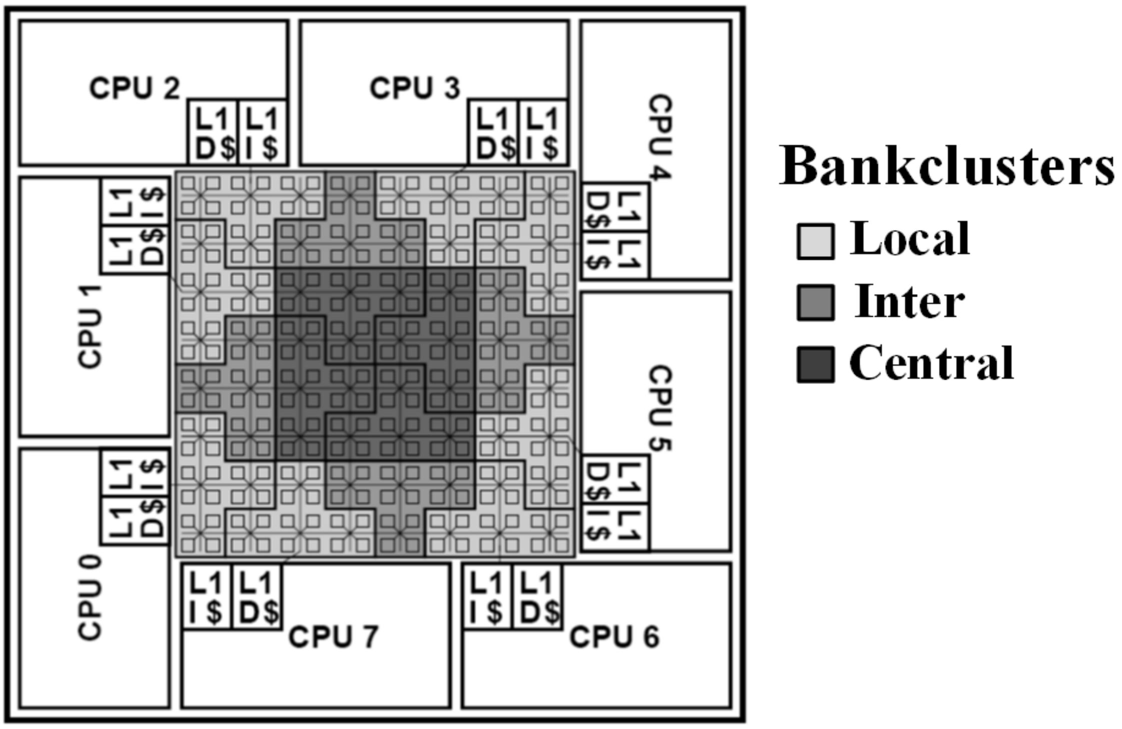


Figure 5.2: Non-Uniform Cache Architecture

The main drawback of this organization for multi-core processor is the long L2 access latency, since it depends on the bank wherein a block is allocated, i.e. a bank in the local bank cluster or a bank in either the central or the local bank clusters of the other cores. The most straightforward way of distributing blocks among the different banks in the non-uniform cache organization is by using a physical mapping policy in which a set of bits in the block address defines the owner bank for each block.

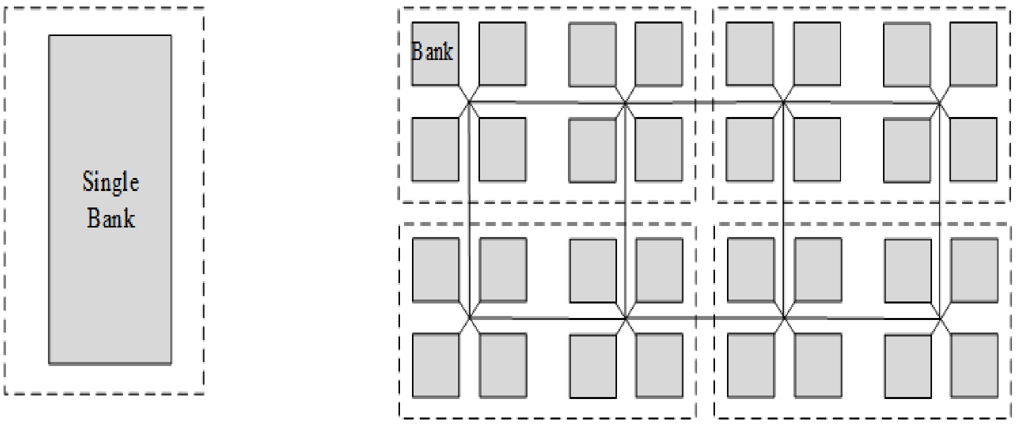
Some recent proposals [63, 134] and commercial CMPs choose the less significant bits for selecting the owner bank. In this way, blocks are assigned to banks in a round-robin fashion with block-size granularity. This random distribution of blocks does not take into account the distance between the requesting core and the home bank on a L1 cache miss. Moreover, the average distance between two cores in the system significantly increases with the increasing number of cores on the CMP, which can become a performance problem for multi-core processors. To address these issues, we proposed a selective cache line replication scheme for shared L2 NUCA. The proposed selective replication mechanism makes use of unused cache lines in the local bank-clusters of different cores. We extend our proposed replication scheme, to balance between access latency and cache capacity in shared NUCA designs by selectively replicating frequently used data close to the requesting cores, while simultaneously ensuring low off-chip memory accesses.

The rest of the chapter is organized as follows: The next section, describes the motivation for this work. Section 5.3 provides detailed explanation of the proposed policy. In section 5.4, the baseline architecture and simulation environment is briefly described, followed by the results and implementation overhead. Related work is discussed in section 5.5 and finally conclusions are given in section 5.6.

5.2 Motivation

In order to adapt to the ever-growing needs of modern memory-hungry work-loads, on-chip cache size need to be increased. Unfortunately, expanding the cache size alone is not sufficient to increase the efficiency since the traditional UCA design exhibits serious limitations as larger capacity comes at the cost of increased access latency. For that reason, large on-chip caches with a single, large and uniform latency are undesirable.

Ideally, we would like data to reside in the part of the cache that is physically located close to the processor so that it can be accessed faster than data that resides farther away from the processor. The solution lies in a distributed cache design that manage to provide varying access times and increased bandwidth. In order to achieve this goal, a complete shift in the cache architecture design paradigm was required. The previously single, monolithic chunk of cache (UCA) is transformed to a finer-grained structure, as shown in Figure 5.3.



(a) UCA (b) NUCA

Number of banks: 1 bank : 32 banks

Average loaded access time: 255 cycles : 24 cycles

Figure 5.3: Shared Level-2 Cache Organization

More specifically, the last-level cache is composed of physically independent *banks*, which are evenly distributed across the die area. This design provides varying access latencies between the cores and the cache banks, depending on the physical distance between the requesting core and the cache bank where the requested data resides. This leads to a Non-Uniform Cache Access (NUCA) organization of the cache. NUCA provides faster access to cache blocks in the banks that reside closer to the processor.

For example, as suggested by Kim et al. [19] and illustrated in Figure 5.3(b), the closest bank in a 16 MB, on-chip L2 cache built in a 50 nm process technology can be accessed in 4 cycles, while an access to the farthest bank might take up to 47 cycles. On the other hand, every access to a UCA of the same size would require a constant latency of 41 cycles. As access time is directly related to the block's placement, the placement is an important decision. Figure 5.3(b) shows a banked NUCA cache, as opposed to the classic UCA shown in Figure 5.3(a). This static NUCA design uses a two-dimensional switched network, permitting a large number of small, fast banks to be interconnected. The NUCA design allows accessing each bank at different speeds, proportional to the distance of the bank from the requesting core. Thus, the closest bank can be accessed in the minimum time, while an access to the farthest is the slowest. A block can only be placed in a single location during its lifetime. This, of course, imposes serious limitations with this architecture: a frequently accessed block may be placed in a bank located far from the requesting core, thus suffering the overhead of a high access time every time it is accessed. The block cannot be placed to any other bank, closer to its requester, in order to improve its access time, since its location in the cache is statically defined by its address.

This limitation of the static NUCA motivated us to propose selective cache line replication in the NUCA cache, which addresses the problems that arise from static placement of cache blocks.

5.3 Proposed Selective Replication Policy

We assume Last level shared L2 Cache as a Non-Uniform Cache Architecture, based on Kim et al.'s NUCA design [16]. The following definitions will help facilitate describing our baseline architecture.

Owner Bank: The bank in which data is mapped for the first time after an off-chip access using the static address mapping scheme.

Bank clusters: A group of eight banks compose a bank-cluster and the complete NUCA cache (128 banks) is divided into a 16 bank-cluster as shown by red dotted box in Figure. 5.4. Each bank cluster consists of a single bank of logical bankset.

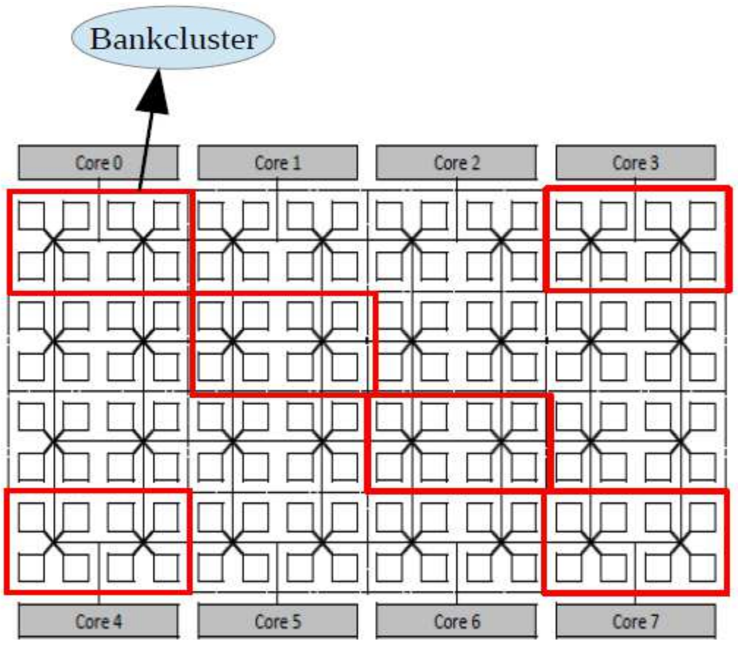


Figure.5.4: Bank cluster in NUCA

Bank set: All the banks that compose NUCA cache can be logically treated as a set-associative structure as shown in Figure. 5.5, where each bank in a bank-cluster holds one way of a logical bank set.

As shown in Figure. 5.5, the complete NUCA cache is partitioned into 128 banks, which is logically organized into a 16-way bankset associative structure (Grey color banks constitute a bankset). Now, the group of eight banks (bankcluster) that are located close to the cores are

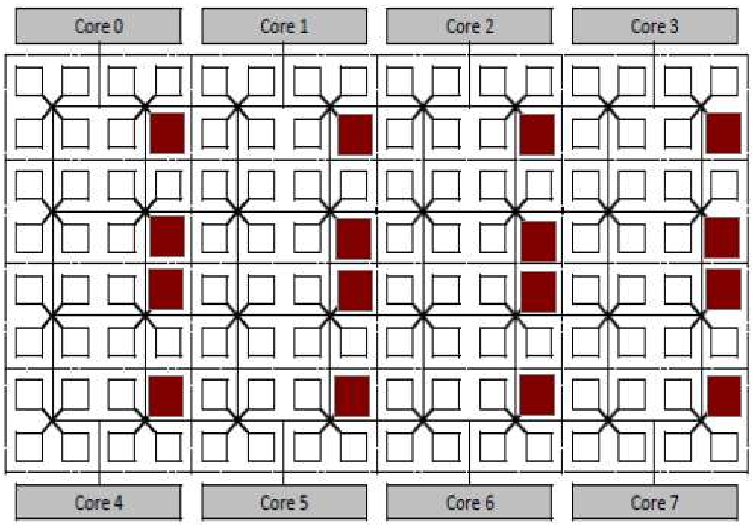


Figure 5.5: Non-Uniform Cache Architecture

called local banks whereas the other eight banks that are located at the center of the NUCA cache are called central banks as shown in Figure. 5.6. Therefore, in a bank-set associative NUCA cache a data block can have 16 possible placements (eight local banks and eight central banks).

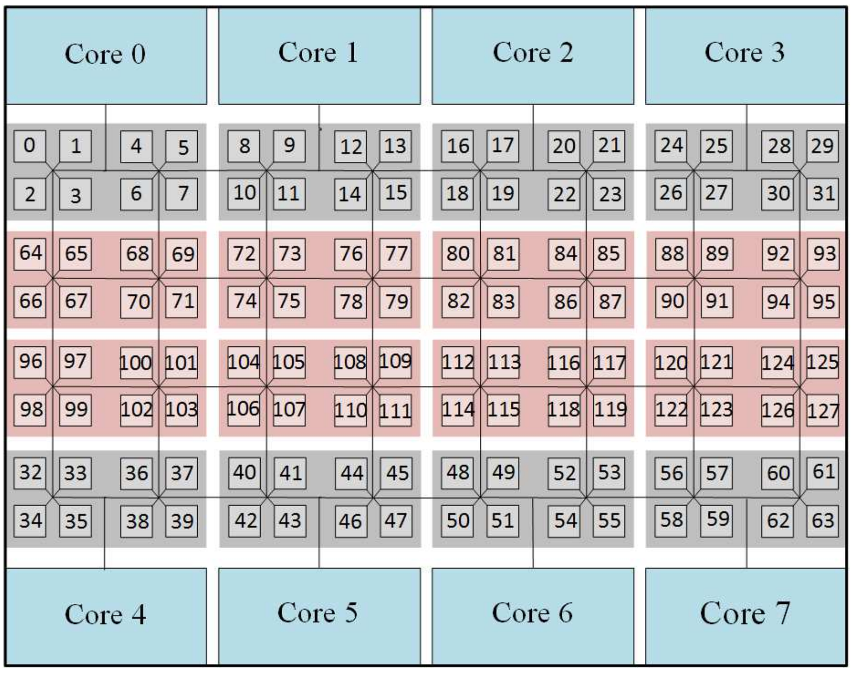


Figure 5.6: Shaded red portion constitutes the central bank-clusters, whereas light brown bank close to the cores are the local bank-clusters

The address mapping of the incoming data block when it comes from off chip main memory is statically predetermined by selecting lower bits of the data block address as shown in Figure. 5.7. The LRU data block in the referenced set of this bank would be evicted if the set is completely occupied by data blocks.

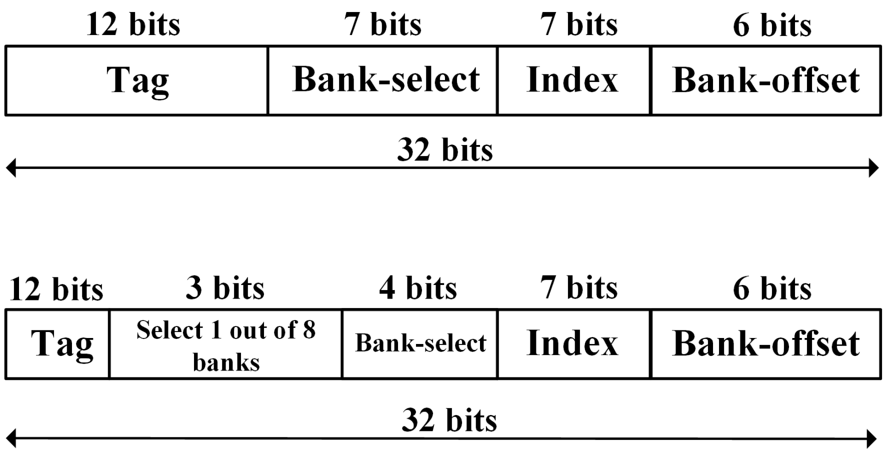


Figure 5.7: Address Interpretation

5.4 Replication Policy: owner bank knows when to replicate

In this section, we propose an efficient, highly accurate and low-overhead mechanism to track the re-usability of each cache line in the shared NUCA. Our scheme allows dynamic replication of those cache lines that shows high usage at the shared LLC. When a replicated cache line is evicted or invalidated, the proposed scheme dynamically adjusts its future replication decision. This scheme also reduces access latency and energy consumption by selectively replicating the cache line that shows high re-usability to the local bank-cluster of the requesting core. It also maintains coherence complexity similar to that of a conventional non-hierarchical coherence protocol as replications are allowed only in the local bank cluster of the requesting core. The extra coherence complexity arises only when the replicated cache line is evicted or invalidated from the local bank-cluster.

5.4.1 Working of the proposed scheme

For proper working of the proposed scheme, we identified four key requirements for efficient cache line replication in the NUCA cache. The first involves selecting a cache line for replication. The second one is the intelligent placement of the replicated cache line. The third requirement is the lookup mechanism capable of quickly locating the replicated cache line within the shared cache and finally maintaining cache coherence for the replicated cache lines. We first define few terms to facilitate describing our proposed scheme.

Owner Bank: The bank where data is placed for the first time, after being brought from off-chip memory. All the subsequent off-chip requests are serialized at this bank for maintaining coherence and resolving false misses.

Copy Sharer: A core that is given access to a separate cache line copy in its local bank cluster.

Non-copy sharer: A core that is acting as a simple sharer of the cache line and has not received a separate copy of the line in its local bank cluster.

Owner bank reuse: The number of times a cache line is accessed at the owner bank before being evicted or written.

Replicated line reuse: The number of times the local copy of the replicated cache line is accessed before it is invalidated or evicted.

Reuse threshold (RCT): If the value of re-usage becomes equal to or greater than this value, then a separate copy of the cache line is created.

Note that for any cache line, one core can be a single copy sharer while other cores can be non-copy sharers of the cache line. So, initially all the cores are non-copy sharers of the cache line as shown in figure 5.8. We have used a directory based coherence protocol, in which each cache entry is further extended with an extra replication indicator bit (RIB) and a 2 bit saturating counter (RCT-1) as shown in the Figure. 5.8. Based on the value of RCT-1 and the status of RIB, the cache controller allows creating a separate copy of cache line in the local bank-cluster.

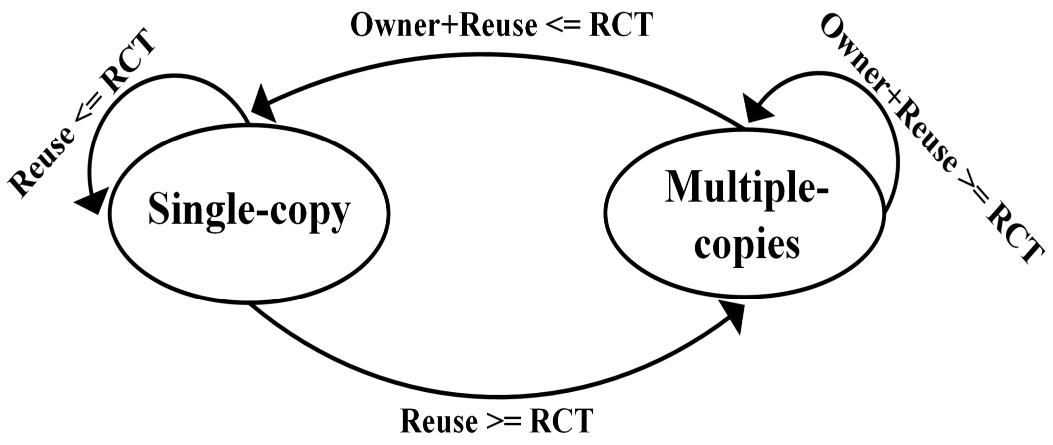


Figure 5.8: State transition based on the value of reuse threshold

5.4.2 Managing Read/Write Request

This section describes how our proposed scheme manages a read/write request and handles evictions and invalidations for replicated cache lines.

5.4.2.1 Read Request

As a result of a compulsory miss, a data block is loaded into the cache from off-chip memory. The cache controllers are designed in such a way that on a L1 cache read miss, it first searches the local bank cluster of the requesting core (where the requested data block can be mapped or replicated within the NUCA cache to provide reduced access latency). If the request hits, the block is inserted at the L1 cache of the requesting core. In addition, if this is the replicated copy of the cache line then the corresponding replication reuse counter (RCT-2) should be incremented to keep track of the line reuse information. In our scheme, for a newly replicated cache line, the counter RCT-1 is reset to 1 and RCT-2 is incremented on every request that results in a hit. Figure 5.9 shows the directory entry to track a replicated cache line. In case of a miss, the memory request is forwarded to the owner bank by using the lower address bits of the block address (Figure. 5.7), beginning the next stage of the search mechanism. If the data block is found, the request is hit and the block is sent to the core that started the memory request, thereby completing the search mechanism. In case the cache line is not found in the owner bank, the memory request is forwarded to off-chip memory.

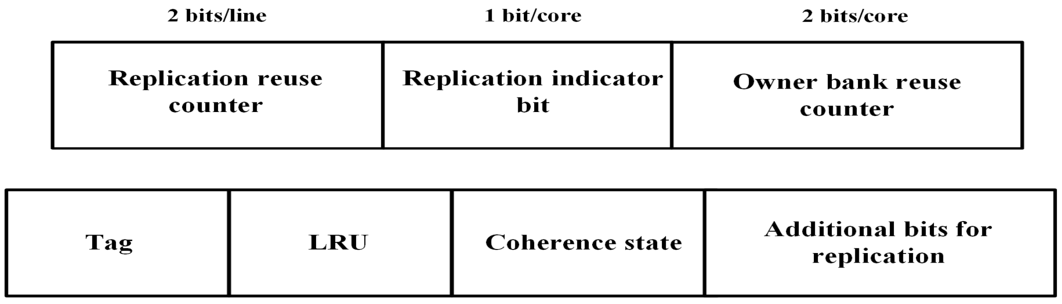

2 bits/line
1 bit/core
2 bits/core
Replication reuse counter
Replication indicator bit
Owner bank reuse counter
Tag
LRU
Coherence state
Additional bits for replication

Figure 5.9: Additional in-line directory bits for the proposed scheme

Algorithm-1, presents how to handle read requests from the cores. The logic for all the algorithms are implemented using full system simulator. To ensure the correct operation and accuracy of our proposed block replication policy, the in-line cache directory entries are extended with extra bits as shown in Figure. 5.9 to keep a track of reuse as well as replicated line information. These additional bits include the Replication indicator bit (RIB), which identifies whether a replicated copy of cache line is created. If it is set to 1, then an extra copy of cache line is placed in local bank-cluster of the requesting core. Secondly, there is a separate owner bank reuse counter (RCT-1) for each core. This counter is used to track the number of times the line is accessed by a core at the owner bank. Initially, it is reset to zero and is incremented on every access to the owner bank. If this counter reaches the reuse threshold (RCT) then RIB is set to 1 and a separate copy of the cache line is placed in the local bank cluster of the requesting core. If the value of RCT-1 is less than the reuse threshold (RCT), then the cache line is inserted in the private L1 cache of the requesting core, without being replicated. In order to better understand algorithm-1 and algorithm-2, let the set C = {C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7} represent the cores as described in the baseline NUCA architecture. Let L1 = {$L1_0$, $L1_1$, $L1_2$, $L1_4$, $L1_5$, $L1_6$, $L1_7$} be their respective private L1 caches. We use BC_{local} and BC_{owner} to refer to the local and owner bankclusters respectively.

Algorithm 1##Read request

```
1:  function handleReadRequest
2:  INPUT:  ReadReq $_j$ from $C_i \in C$
3:  Begin:
4:  Lookup  $L1_i$
5:  if (hit)
6:  Load Line $_j$
7:  $LRUQueue_{set}$ .movetoEnd($Line_j$)
8:  else
9:  Fwd ReadReq j $\rightarrow BC_{local}$
10: if (hit)
11: Load $Line_j$
12: $LRUQueue_{set}$ .movetoEnd($Line_j$)
13: RCT-2 ++
14: else // local bank-cluster miss
15: Fwd ReadReq $_j$ $\rightarrow$ $BC_{owner}$
16: if (hit)
17: if (RCT-1 > RCT)
18: RIB $\leftarrow$ 1
19: endif   // Line 17
20: if (RIB == 1)
21: $BC_{local}$.insertReplica (Line $_j$)
22: RCT-1 $\leftarrow$ 1, Load Line $_j$
23: else// RIB!= 1
24: RCT-1 ++, Load $Line_j$
25: $LRUQueue_{set}$ .movetoEnd(Line $_j$)
26: endif            // Line 20
27: else        //owner bank miss
28: Fwd ReadReq $_j$ $\rightarrow$ off-chip
29: endif             // Line 16
30: endif               // Line 10
31: endif                // Line 5
32: End// Line 3
```

Also assumed is a LRU based replacement policy, implemented using a queue. In our analysis, we have considered few special cases that could further accelerate our proposed policy. For example, during the initial search into the local-bank cluster closer to the core, it is possible for the same bank to be the owner bank and the read request can be handled directly at the local bank cluster of the core, resulting in reduced number of steps. In this case, even if the replication indicator bit (RIB) is set to 1 (to create a copy of replicated line) the cache line is only inserted at its private L1 cache, without being replicated.

5.4.2.2 Write Request

In this section, the details of write requests handled by our proposed scheme are presented. In case of a write request, the cache controller first checks the private L1 cache. If it is not present in exclusive state it results in a miss and the local bank cluster is probed for the replicated cache line. If the replicated cache line exists in the exclusive or modified state, it is moved to the private L1 cache and its reuse counter is incremented. If the replicated cache line is present in the shared state or if it does not exist, then the request is forwarded to the owner bank depending on the lower bits of the requesting address as discussed in Figure. 5.7. Upon receiving the request, the owner bank checks the directory information for that line and sends invalidation messages to all other sharers and L1 copies to maintain the single-writer and multiple reader case, thereby simplifying the coherence protocol complexity. Once the invalidation acknowledgements are received, the owner reuse counter (RCT-1) of all the non-copy sharers are reset to 0 except for the requesting core since they have not shown enough cache line reuse. If the requesting core is the only sharer then its owner reuse counter (RCT-1) is incremented otherwise it is reset to 1.

Algorithm-2, illustrates how to handle write requests for the cache line.

Algorithm 2 ## Write request

```
1:    function handleWriteRequest
2:    INPUT: WriteReq $_j$ from $C_i \in C$
3:    Begin:
4:    Lookup $L1_i$
5:    if (hit)
6:    Write Line $_{j,}$ update LRU state
7:    else //miss
8:    Fwd WriteReq $_j$ $\rightarrow$ BC $_{local}$
9:    if (hit && cacheLine $_j$.state == M/EX)
10:   $L1_i$.insert (Line $_j$)
11:   RCT-2 ++
12:   Write Line $_{j,}$ update LRU state
```

```
13:  elsif (hit && cacheLine_j.state == S)
14:  Fwd WriteReq_j → BC_owner
15:  Send Inv. → L1 copies, copy-sharers
16:  RCT-1_other sharers ← 0
17:  Recv. Inv. Ack
18:  if (C_i.isCopySharer(Line_j))
19:  Send RCT-2→BC_owner
20.  Decide Replica status C_i
21:  endif            // Line 18
22:  cacheLine_j.state ← EX
23:  L1_i.insert (Line_j), Write Line_j
24:  update LRU state
25:  if (C_i.isSingleSharer (Line_j))
26:  RCT-1 ++
27:  else
28:  RCT-1 ←1
29:  endif          // Line 25
30:  endif            // Line 9
31:  endif           // Line 5
32:  End             // Line 3
```

5.4.2.3 Invalidation Request

In case of an invalidation request, if a copy of cache line is found in either of the caches (L1 or local bank-cluster), an acknowledgement is sent to the owner bank. If a replicated cache line exists then the replica reuse counter is communicated back with acknowledgement. This information is used to decide whether the core will maintain replica status or not. If the value of RCT-1+RCT-2 (owner reuse + replicated line reuse) is greater than threshold value then it maintains replica status, otherwise it is demoted to the status of a non-copy sharer.

5.4.2.4 Eviction Request

On an L1 cache line eviction request, the local bank-cluster is probed for the same address. If a replicated block exists, then the dirty data in the L1 cache is merged with it, otherwise an acknowledgment is sent to the LLC owner bank. In case the replicated cache line in the local bank cluster is evicted then the L1 cache is searched for the same address and invalidated. An acknowledgment is sent to owner bank with the replicated line reuse counter information. If RCT-2(reuse counter) >=RCT, then the core maintains copy status, otherwise it is demoted to non-copy status.

5.5 Hardware Overhead of Proposed Policy

The proposed replication policy requires additional hardware to implement the selective replication of blocks within the shared LLC. As shown in Figure. 5.9, each directory entry requires 2 bits for the replicated line reuse counter (RCT-2) (for an optimal threshold of 4) and 1 extra bit to store replication information (RIB). Hence, the proposed scheme requires an additional (8X3) + (2X8) X 8 = 152 bits of storage per LLC directory entry. Therefore, the extra number of bits required per bank is 128X152 = 2.375kB. So, as per our baseline configuration with 8 MB LLC NUCA cache consisting of 128 banks, the total hardware required by the proposed scheme is 51.968kB, which is 0.634% additional hardware required. The proposed selective replication scheme can be easily extended to tiled CMPs as well and is not restricted to NUCA based designs. In addition to the hardware overhead, there is additional complexity in cache design partly because of the additional latency introduced by comparison with the threshold, which is taken care of in our design.

5.6 Cache Coherence Protocol

Our work uses a directory protocol that does not need an ordered interconnect to satisfy coherency. We also believe that future CMPs will rely on a directory like structure to maintain coherence and can scale to a large number of on-chip cores. To ensure correctness and to implement different read and write scenarios, cache coherence protocols utilize transition states. Transient states usually include states where the controller is waiting for acknowledgements or data to be received. Our protocol implementation inherits such transition states from the baseline cache coherence protocol and uses these transient states to maintain a coherent view of the system. In the proposed cache access scheme, for any cache line that does not exhibit complex sharing and therefore search mechanism, the implemented protocol works similar to the baseline cache coherence protocol. This is basically enforcing a write-invalidate policy for all cache lines in the shared NUCA. The coherence protocol is designed on top of the write-invalidate directory protocol, which is a modified baseline MOESI protocol. Race conditions are handled using busy or active states for each request. Sequence diagram in Figure. 5.10, briefly describes how a write-invalidate based protocol works for a simple cache line replication example, and the sequence diagram in Figure. 5.11, describes block invalidation. The arrows represent a specific location in the system with a hypothetical time line. From left to right, these locations are the requesting core, the L2 shared cache which also includes the directory that is co-located, the consumer cores, and the main memory.

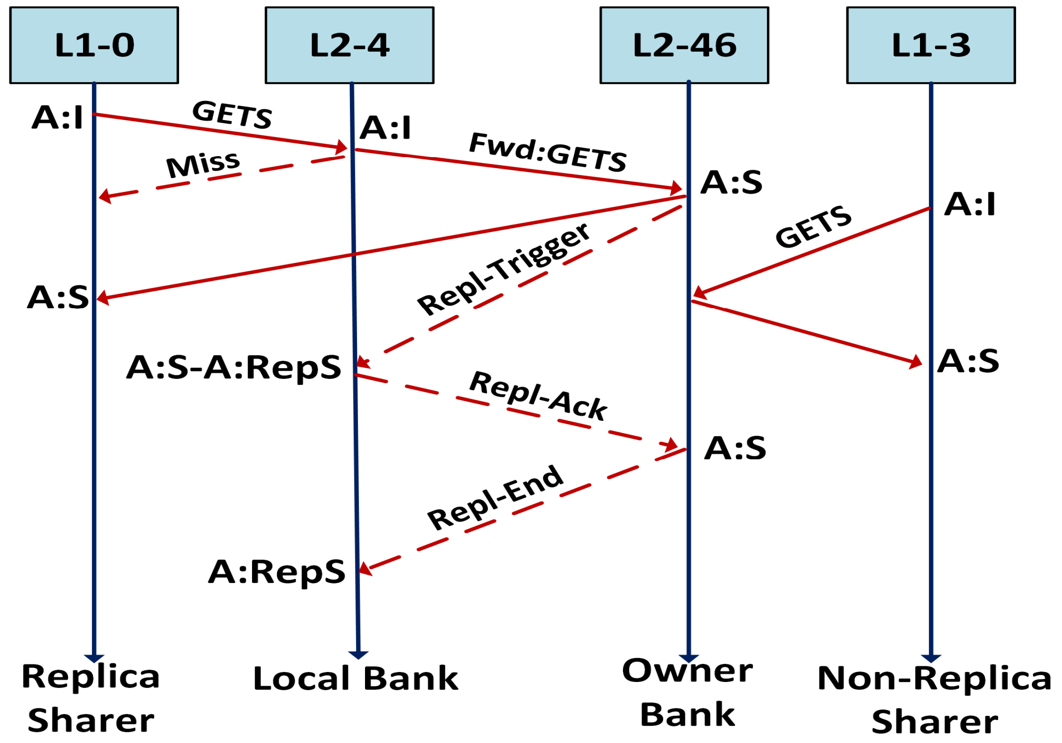


Figure 5.10: Sequence diagram for block replication

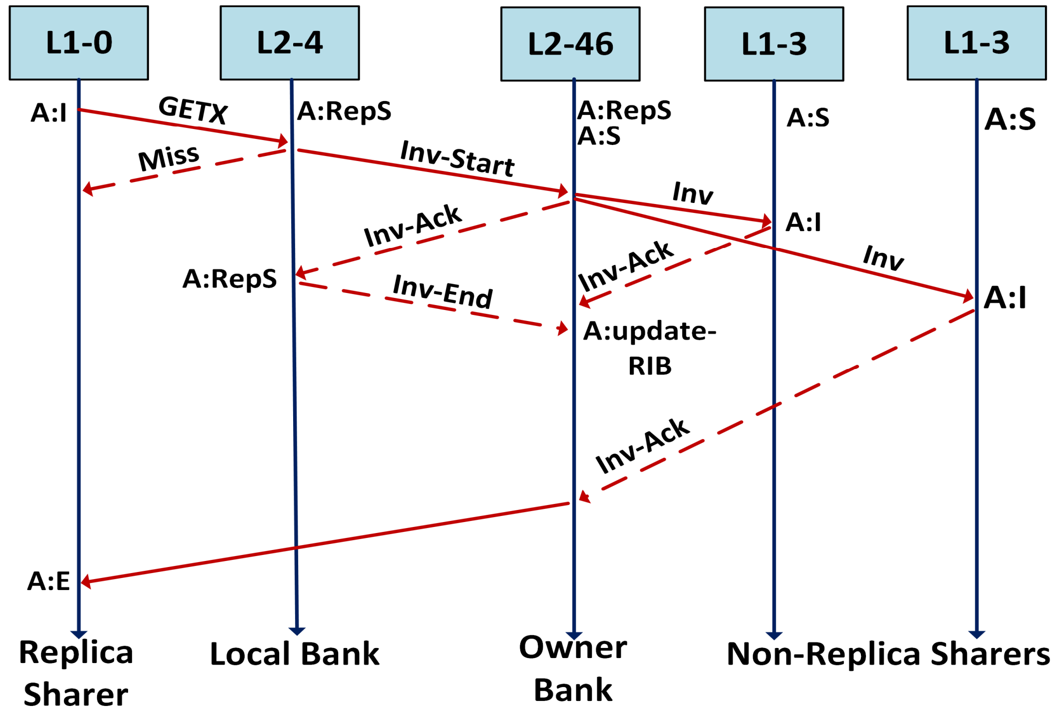


Figure 5.11: Sequence diagram for block invalidation

For clarity in explanation, the example assumes a single requesting core and a single consumer core of the cache line. Also, we assume that initially the cache line is in the

OWNED state in the requestor’s cache and SHARED in the consumer core’s cache. The directory is co-located with each cache line and it tracks the coherence state of cache lines belonging to different cores.

5.7 Verification of Protocol

Modified MOESI based directory protocol relies on the baseline coherence protocol for correctness. However, before the protocol is put into operation, it is essential to verify its robustness when subjected to different race scenarios. A robust coherence protocol is required to ensure correctness under all possible conditions. For the verification, we have utilized the stress tests provided by the GEMS toolset. By stress testing over a large design space encompassing all possible race conditions, certain coherency issues were identified and the protocol was suitably modified and corrected.

5.8 Experimental Setup

5.8.1 Simulation Environment:

In this section, we describe our evaluation methodology with all the results obtained using the system configuration described in Table 5.1. We simulated the entire system using Virtutech Simics full-system simulator [99] extended with the GEMS toolset [100]. GEMS is an event driven simulator that provides a complete memory-system timing model that enabled us to model the multi-banked NUCA cache architecture.

Table 5.1: System Configuration

Configuration Parameters	
No of Cores	8
Core Mode	Single Thread
Frequency	1Ghz
L1-Data Cache	32kb, 64 bytes
L1-Instruction Cache	32kb, 64 bytes
Shared L2 Cache	8 Mb, 128 banks
Bank Size	64 Kb, 8-Way, 64bytes

Furthermore, the RUBY memory system simulator provides support to implement baseline system memory hierarchy. This includes the on chip interconnection network parameters, bank access time, mapping, replacement policies etc. In RUBY, each cache bank has its own controller and using the domain specific language called SLICC we can specify with precision the coherence protocol. This environment allows us to simulate a complete multiprocessor system that is running a commercial operating system without any modification and it accurately models the network contention introduced during the simulation. The simulated system is organized as a single CMP that consists of eight UltraSPARC IIIi homogeneous cores with layout depicted in Figure 5.4. Each processor core has its own first-level cache (data and instructions) and is connected to a node of the network. The last level of the memory hierarchy is the NUCA distributed in 128 banks connected to the cores via switches. We used MOESI based directory protocol to maintain correctness and robustness of the memory subsystem. The main system configuration parameters used in our simulations are shown in Table 5.2.

5.8.2 Benchmarks

To quantitatively analyze the proposed scheme, we used two different scenarios: 1) Multi-programmed and 2) Parallel applications. The first one executes in parallel a set of eight different SPEC CPU2006 workloads with the reference input and fast forwarded to the beginning of the main loops. Table 5.2 outlines the workloads that make up this scenario. The Parallel workload simulates the complete set of applications from the PARSEC v2.0 benchmark suite [108] with the sim-large input data sets. This benchmark suite contains 13 programs from different areas such as, computer vision, image processing, financial analytics, video encoding and animation physics. The method for the simulations involves first skipping both the initialization and thread creation phases and then fast-forwarding while warming up the cache for 500 million cycles.

Table 5.2: Benchmarks

Benchmarks	Applications	Input
PARSEC	Blackscholes, bodytrack, canneal, racesim, fluidanimate, x264, raytrace,swaptions, streamcluster	Sim-large Input
SPEC2006	Mix or Different applications, gcc, ibm, astar, mcf, soplex, perlbench	Reference Input

Then finally, we performed a detailed simulation for 500 million cycles. We use the aggregate number of user instructions committed per cycle as the performance metric, which is proportional to the overall system throughput.

5.9 Results

We have simulated the execution of selected applications from the PARSEC multithreaded benchmark suite [108] to completion using sim-large input set. We have used energy consumption of the shared cache memory and the completion time as the reference performance metrics. We have also analyzed the network traffic in terms of the bytes-per-instruction and L2 hit latency to further evaluate our proposal. For the applications that have high miss rate in the NUCA cache, our scheme outperforms the S-NUCA baseline architecture by 8% as shown in Figure 5.12. By taking advantage of selective replication for highly reused cache lines at the owner bank, memory requests are directly satisfied by only accessing the local bankcluster. The percentage of improvement is obtained by taking the difference between the average value along all the applications for reference and proposed schemes. Figure. 5.12, shows the normalized completion time for the selected benchmarks and we observe that in none of the considered benchmark applications, performance is degraded.

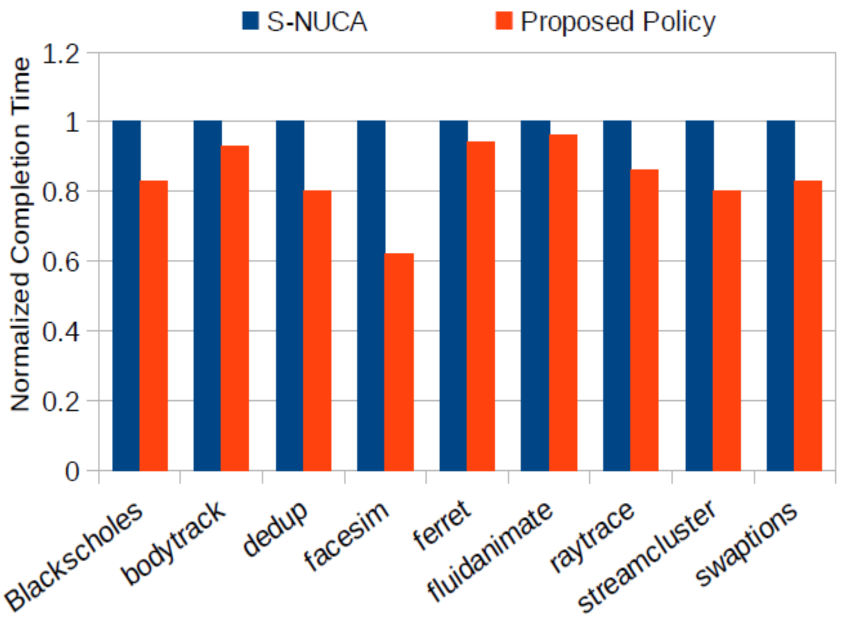


Figure 5.12: Normalized Completion Time

In the chosen applications, completion time reduction varies from about 4% up to 36%. On average, this translates to about 8% increase in performance. Figure. 5.13, shows the average L2 hit latency in both S-NUCA and the selective replication scheme. With the adoption of the replication mechanism, L2 response time reduces by close to 12% on average; this is due to the fact that most of the hits are concentrated in the faster local-banks and the requested blocks can be provided in a very short time. For few applications like Streamcluster and Bodytrack, we have observed low L1 miss rate, so they can't gain much from the proposed policy but there is no further degradation in their performance. Therefore, for the applications with higher miss rates, the impact on the performance is even better. In the second scenario, we have observed applications with low high miss rate, like Dedup and Swaptions. In this scenario both the schemes take equal access latencies when the request hits in the closest banks. With applications having very high hit rate like Bodytrack, we have observed slight performance improvement. We assume that the applications running on future processors will follow the characteristics of the first scenario: applications with large working sets and many applications running simultaneously.

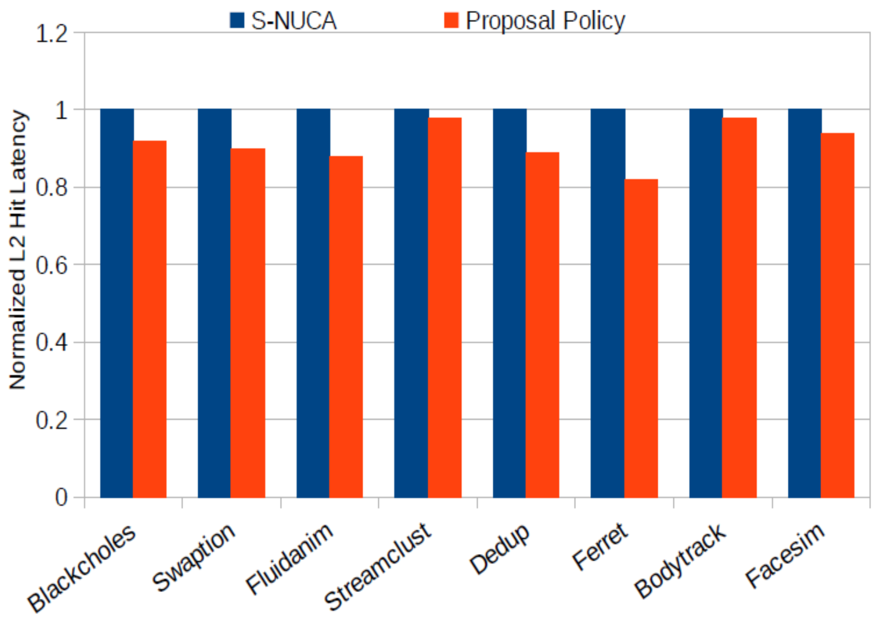


Figure 5.13: L2 Hit Latency

In general, our scheme shows good performance improvement with almost all benchmarks of the PARSEC suite, with more than 8-10% improvement in the Ferret application. Figure. 5.14, presents the distribution of the data as well as control messages that affect the overall network traffic in terms of number of bytes per instruction. In our architecture the size of control message is 8 bytes (header only) whereas the size of the data message is 72 bytes which contains 8 bytes for the header portion and 64 bytes for the data block. From Figure. 5.14, it can be observed that the total network traffic is reduced for almost all the applications which are the result of selective replication of cache lines at the closer banks (local bankclusters). The closer banks are the banks in the local bank-cluster that are physically close to the cores. This reduces the network distance traversed by a packet to reach the receiver bank. In the proposed policy the data packets traverse a lesser number of hops with respect to S-NUCA, as seen in the graph and the data portion of the network traffic is reduced whereas the control part of the network traffic remains almost the same. In our proposed policy, the selective replication and then invalidation for read-write blocks is triggered only few times as compared to the total number of L2 accesses.

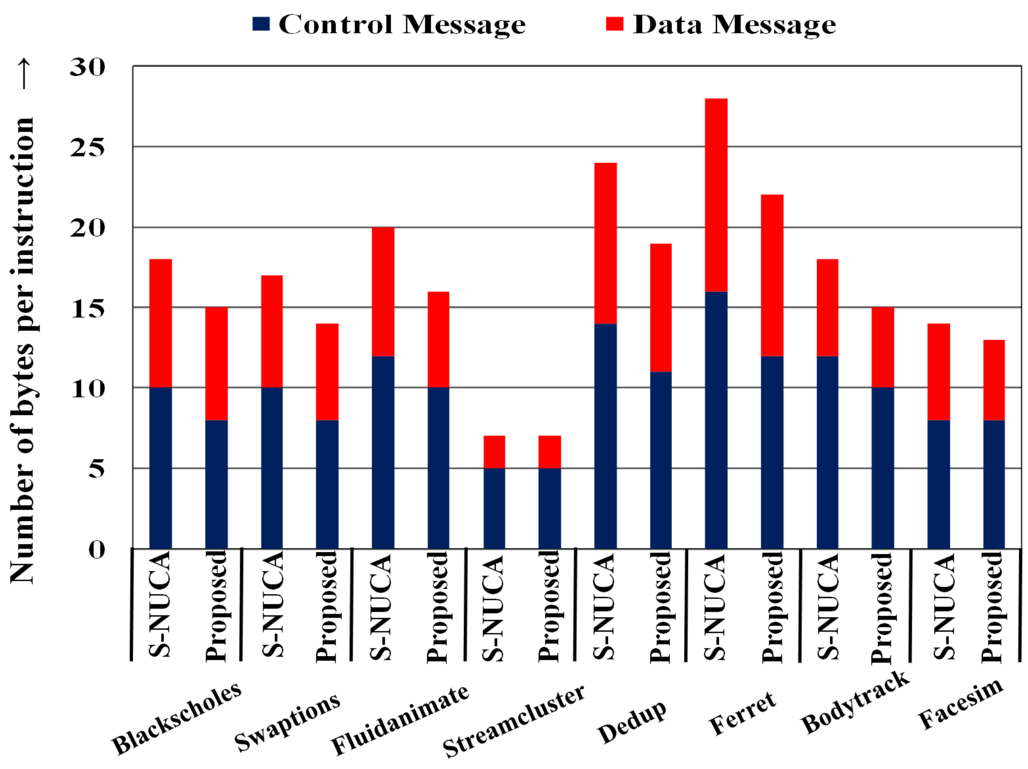


Figure 5.14: Distribution of Network Traffic

As a result, the overhead of replication and invalidation messages has a low impact on the total network traffic. Reduction in network traffic as shown in Figure. 5.14 reduces dynamic energy consumption because of reduced overall network activity. Figure. 5.15, shows the dynamic energy consumption of each benchmark using the proposed selective replication policy.

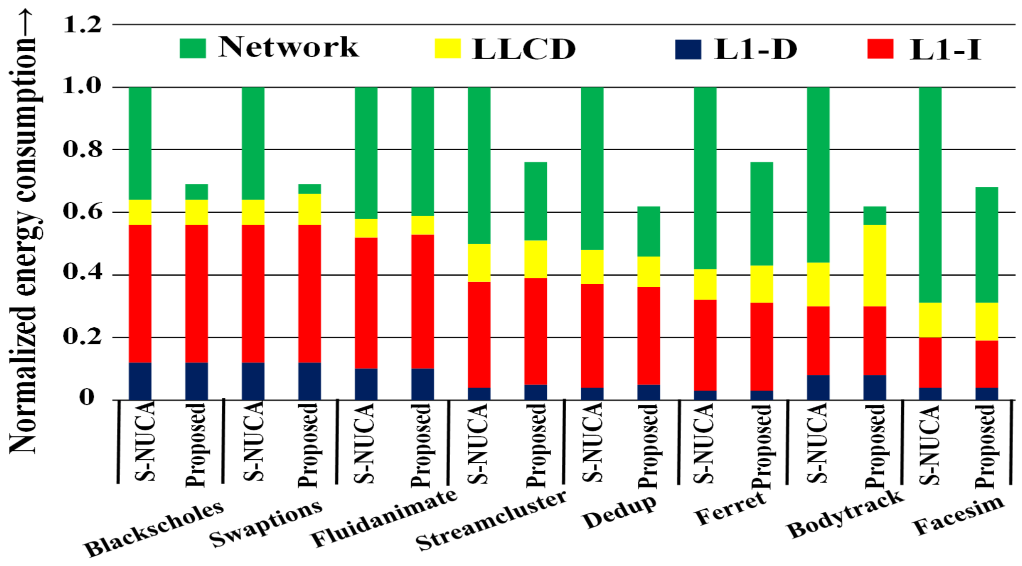


Figure 5.15: Normalized Energy Consumption

The energy reduction can be primarily attributed to the reduction in network traffic. Therefore, for benchmarks where our proposal improves the L2 performance, the energy benefits will in fact be higher. We observed that the proposed scheme improves energy consumption of the NUCA cache by more than 27% as compared to the S-NUCA baseline architecture. To summarize, the proposed selective replication policy reduces energy consumption and enhances performance when compared to other last level shared NUCA data management schemes. We explored all values of RCT between 1 and 8 and found that they provide no additional insight beyond a threshold value of 4. The proposed policy makes use of data locality on-chip and reduces off-chip miss rate. Overall, our replication policy consumes 27%lower energy and shows 15% lower completion time when compared to S-NUCA.

5.10 Related Work

Prior research on cache management in multi-core processors has mostly focused on the last level cache. Shared, private as well as hybrid LLC designs have been extensively reported in [112] [113] [120] [130] [132]. All other cache levels have traditionally been organized as private to a core. Private LLC organizations provide limited cache capacity to a thread and adversely affect applications with large working sets. Shared organizations on the other hand have the flexibility of storing the data of an application in various locations throughout the cache, but at the cost of higher hit latencies since each request has to incur the wire delays imposed by the meshed interconnection network. However their off-chip miss rates are low as compared to private organization because data is not replicated in the LLC. The influence of wire delay in shared LLC design implies that access latencies are not constant. To address this problem of non-uniform access latencies, Kim et al. [19] introduced the original non uniform cache architecture (NUCA) as shown in Figure. 5.2. In shared NUCA, the whole LLC is partitioned into smaller banks such that nearer cache banks have lower access latencies as compared to farther banks, thus mitigating the effects of on chip wire-delays. The efficiency of a migration scheme depends on an accurate data access scheme that was difficult to implement in the past. Kim was the first to highlight the importance of the bank access scheme in dynamic NUCA organizations. Although block migration enhances D-NUCA benefits to outperform S-NUCA, it is limited by the quality of the bank access scheme within NUCA. This work was further extended by Huh et al. [63] who analyzed different NUCA organizations and came to the same conclusion that although D-NUCA outperforms other organizations, access policy is of prime importance in shared D-NUCA designs. Since then researchers from both industry and academia have extensively studied policies in NUCA architectures that efficiently manage: block placement [63][115][127][129][131], block migration [126][127][135],replacement [128] and lookup [125][137]. The introduction of CMPs further increased the complexity of the multi-banked NUCA design process. Chisti et al. [134] also proposed an alternative NUCA design called NuRAPID, in which the Last level cache is divided into a few large banks instead of many smaller banks for higher reliability, efficiency and lower data migration rates with further extension to accommodate a limited number of cores. The concept of cooperative caching in multi-core processor systems was introduced by Chang et al. [114] to increase the overall cache capacity, where each processor core has a local L2 cache and cache consistency, sharing are achieved by listening in on all the L2 cache traffic and cooperating in decreasing the conflicts. Another variant of NUCA is proposed by Liu et al. Beckmann and Wood [112] in their analysis show that block migration policy is less effective for CMP because 40-60% of total hits in commercial workloads were satisfied in the central banks. There has also been significant recent work in evaluating the benefits and limitations of replication in CMP caches. Huh et al. [63] investigated sharing in a CMP-NUCA cache and favored some replication between cache banks. Adaptive Selective Replication dynamically evaluates the costs and benefits of replication on a per-block basis and adapts to the needs of the workload. Other schemes similar to Adaptive Selective Replication are the CMP-NuRAPID [134] and Cooperative Caching [114] proposals. The above proposals reduce replication but utilize a static mechanism that does not adapt to the needs of different workloads in different phases and other constraints. Finally, similar to ASR, Suh et al. [138] used set and way counters to monitor cache block utilization. Zhang et al. [116] used an automatically re-sizable cache with a miss tag buffer to track possible cache hits if a full sized cache was available. However, Suh et al. used the monitoring information to dynamically partition the ways in a set-associative cache among multiple thread sand Zhang et al. used it to reduce energy consumption.

5.11 Summary

We have proposed an efficient selective replication policy for the last level cache. The cache line re-usability is profiled dynamically using in-directory reuse counters. On a set of multi-threaded applications, our selective replication policy reduces the overall energy by 27% and the completion time by 15% when compared with Static-NUCA L2 cache management policy. The coherence complexity of our protocol is almost identical to that of a traditional non-hierarchical (flat) coherence protocol since replicas are only allowed to be created at the LLC slice of the requesting core. Our proposed policy is implemented with an extra storage overhead of 3.71% per NUCA bank.

Chapter 6

Adaptive block Migration-Replication (AMR) in NUCA

This chapter presents challenges introduced by dynamic features provided by D-NUCA, like multiple locations for data placement, migration movements and data access policy. This chapter presents AMR scheme, which is an efficient and cost-effective mechanism to overcome above challenges and reduce miss latency in the NUCA cache and the on-chip network contention.

Chapter 6

Adaptive block Migration-Replication (AMR) in NUCA

6.1 Introduction

The static non-uniform cache architecture (NUCA) designs for shared last level cache memory outperforms the classical uniform cache organization with slightly increased complexity in the control mechanism. In S-NUCA as shown in Figure 6.1 (a) and (b), a block can only be placed in a single location. This, imposes serious limitations with this architecture: a frequently accessed block may be placed in a bank located far from the core, thus suffering the overhead of a high access time every time it is accessed. The block cannot be placed to any other bank, closer to its requester, since its location in the cache is statically defined by its address. This limitation of the static NUCA motivated us to propose selective cache line replacement in the NUCA cache, which address the problems that arise from static placement of cache blocks in the previous chapter.

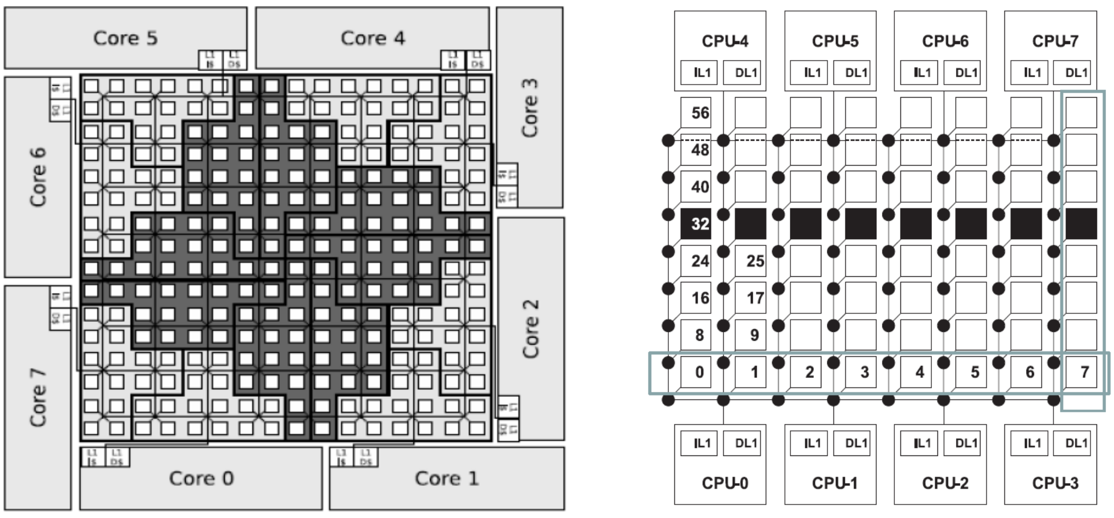


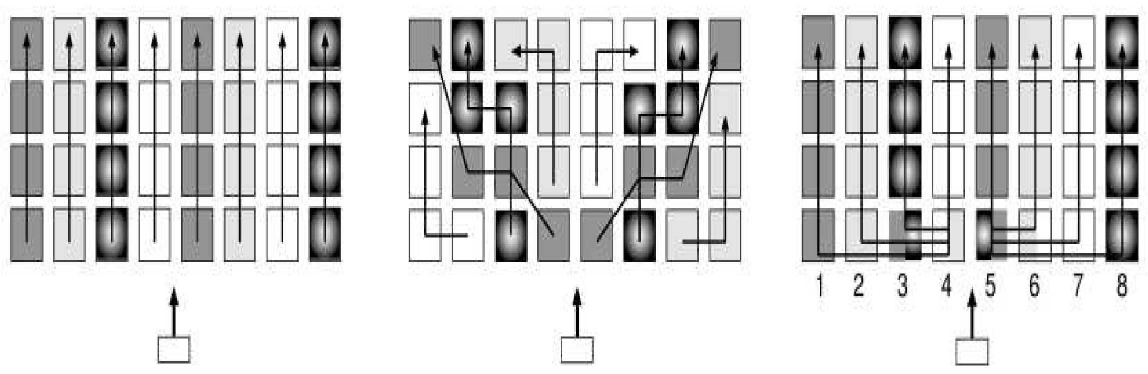
Figure 6.1: NUCA Organizations

However, the limitations of the static NUCA organization gave birth to NUCA's next generation designs, the dynamic NUCA, which address the problems that arise from static placement. Data movement and their management further impacts memory access latency and consumes power. We observed that previous D-NUCA designs have used a costly data access scheme to locate data in the NUCA cache in order to achieve remarkable performance improvement. To address these issues, we further investigated this limitation along with the benefits of dynamic NUCA organization as well as discusses the drawbacks of both S-NUCA and D-NUCA organization. Finally, we proposed an adaptive migration-replication policy for non-uniform shared L2 cache supported with an efficient data access policy using a set of location pointers with each banks, which provides solutions to the basic problems with D-NUCA. Our scheme relies on low-overhead and highly accurate in-hardware pointers to control network traffic and improves cache miss latency. Using simulations for 8-core multi-core system, we show that our proposed data search mechanism in D-NUCA design reduces 40% dynamic energy consumed per memory request and improves average performance speedup by 6%.

The rest of the chapter is organized as follows: The next section describes motivation for this work. Section 6.3 provides detailed explanation of proposed data access policy. In section 6.4, the baseline architecture and simulation environment is briefly described, followed by the results and implementation overhead. Related work is discussed in section 6.5 and finally conclusions are given in section 6.6.

6.2 Motivations for This Work

As technology nodes evolve, feature sizes keep shrinking with every generation. However interconnects have scaled by a much smaller amount. Hence wire delays have shown slight improvements and have now become a major hurdle in improving chip multiprocessor (CMP) performance. This discrepancy has led to an increased focus in developing on-chip cache architectures that can minimize the increasing wire delays [19] [63] [5] [138]. With increasing number of cores physically distributed on-chip, accesses from different cores incur non-uniform delays. Such an observation has led to the development of heavily banked non-uniform cache architectures (NUCA), with an aim to utilize the closer banks to satisfy the requests of different cores. Figure 6.2 shows three different ways of assigning sets to banks proposed by Kim [19]. The migration mechanism proposed for these mapping schemes was fairly simple, since it is tightly related to the organization of the banks in sets. When a hit occurs to a data block in one of the cache's banks, it is swapped with the corresponding block of another bank that belongs in the same bankset and is one step closer to the cache controller.



(a) Simple Mapping (b) Fair Mapping (c) Shared Mapping

Figure 6.2: Set sharing schemes in NUCA Organization with different mapping schemes [16]

Traditionally, NUCA organizations have been classified as static (S-NUCA) and dynamic (D-NUCA). While in S-NUCA a data block is mapped to a unique bank in the NUCA cache, D-NUCA allows a data block to be mapped to multiple banks. D-NUCA also provides dynamic features like migration of data between multiple banks by leveraging data locality and moves frequently accessed data close to the requesting core. Multiple placement locations for data and its migration between multiple banks, makes the data access scheme a key constraint in D-NUCA based architectures. However, because of non-uniform distances between requesting cores and shared L2 cache banks in the mesh interconnection network, on chip cache access latencies vary greatly and can sometimes be very large due to wire delays. Extensive research has been reported in literature dealing with such non uniform cache architectures (NUCA) [19] [63] [110] [125] [129]. Several replication mechanisms have also been proposed to balance between access latency and cache capacity in hybrid L2 cache designs [60] [139]. Much of the previous efforts have focused on either the migration or replication of blocks in the shared last level cache [60] [127]. Such a “one-policy-fits-all” approach may adversely affect some applications that show greater benefit from using one policy over the other. In this work, we proposed an adaptive migration-replication scheme that is tuned to the varying runtime requirements of an application. The proposed scheme analyzes the access patterns of applications during their execution in order to make the migration/replication decision. Our approach is adaptive in the sense that it can shift between the two policies (migration and replication) at runtime in order to best suit the requirements of the application. Methods that have implemented a selective cache line replication scheme on top of a migratory baseline policy lack an effective search mechanism to make best use of the low access latency provided by replicated lines. To alleviate this drawback, we propose an effective search policy to keep track of cache lines in the shared LLC. We have also explored several exceptional cases that may arise because of replica creation and the race conditions that it may cause, if left unsolved. The baseline coherence protocol was suitably modified to ensure coherency of data in all possible scenarios.

In summary, we propose a novel runtime shared cache management scheme that uses both an accurate, low overhead data access policy and an adaptive migration-replication mechanism to meet the performance requirements of different applications in different phases of their execution.

Following are our contributions in this effort:

1. Dynamically adapting the migration/replication decision at runtime according to the needs of a particular application.
2. Proposing an accurate, low overhead data lookup policy that provides low latency cache access in the presence of both replicated as well as migrated blocks in the cache.
3. Identifying possible race conditions that may arise due to the presence of both migrated and replicated blocks in the cache and appropriately modifying the baseline coherence protocol to handle these exceptional scenarios.

6.2.1 Exploiting Dynamic Non Uniform Cache Architecture

The migration mechanism allows data to move towards the most frequently referring core, thus reducing the average cache latency by storing the most frequently accessed blocks in banks close to the referring core. In CMP configurations in which processors are placed at different sides of the shared D-NUCA cache, the performance improvements due to the migration can be limited by the ping-pong or conflict hit phenomenon [110] [103] [60] shown in Figure. 6.3. We recall that the typical way to implement migration consists in letting cached blocks to migrate whenever a request coming from an L1 cache hits the block, and letting them migrate in the direction of the requester to decrease access time. When the requests for the same cache block are generated by L1 caches staying at opposite sides of the D-NUCA (e.g., L1 from CPU-0 and L1 from CPU-4 in Figure. 6.1(b), the blocks alternatively migrate up and down in the pertaining bankset, usually staying in the middle of the bankset, that is, far from all the processors. The effects of ping-pong are twofold: First, the performance improvements due to migration are limited, as shared blocks don't succeed in reaching the faster ways and secondly the dynamic energy consumption increases, due to the increased NOC traffic induced by up and down migration of blocks.

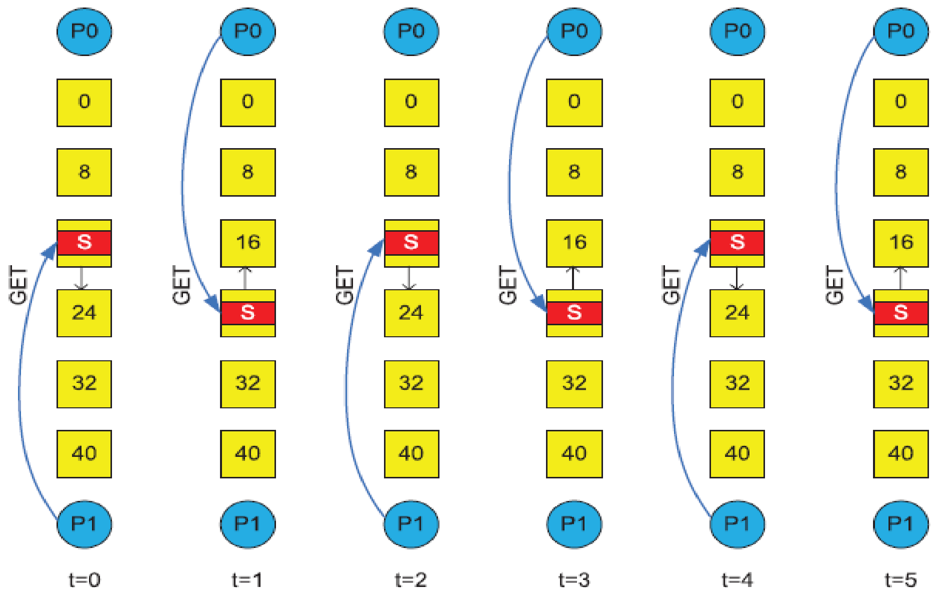


Figure 6.3: Data ping-ponging between banks 16 and 24, and it is not able to reach near the local bank clusters in Dynamic NUCA Organization

6.2.2 Data Lookup with in the D-NUCA

A NUCA design can be characterized based on four policies which determine its behavior:

Bank placement, which determines the first location of data in the cache.

Bank lookup, which defines the searching algorithm across the banks.

Bank migration, which decides data movements between the NUCA's banks.

Bank replacement, which deals with the evicted data and any actions required upon its eviction.

Static NUCA implements static placement of data (standard placement depending on its address), which also allows a simple static lookup mechanism, using the same static function that is used for placement. It also implements a classic replacement policy, e.g. LRU, and no migration of data at all. A data block is placed in a predetermined, statically determined by its address, position and never moves until evicted. At the other extreme, in a dynamic NUCA, a data block can be placed in any bank of the cache. This approach provides the greatest flexibility and unlocks the possibility for greater performance gains. However, such an extremely dynamic placement strategy comes at a cost. The overhead of locating a data block in the cache when it could be found anywhere, can be too large as shown in Figure 6.4.

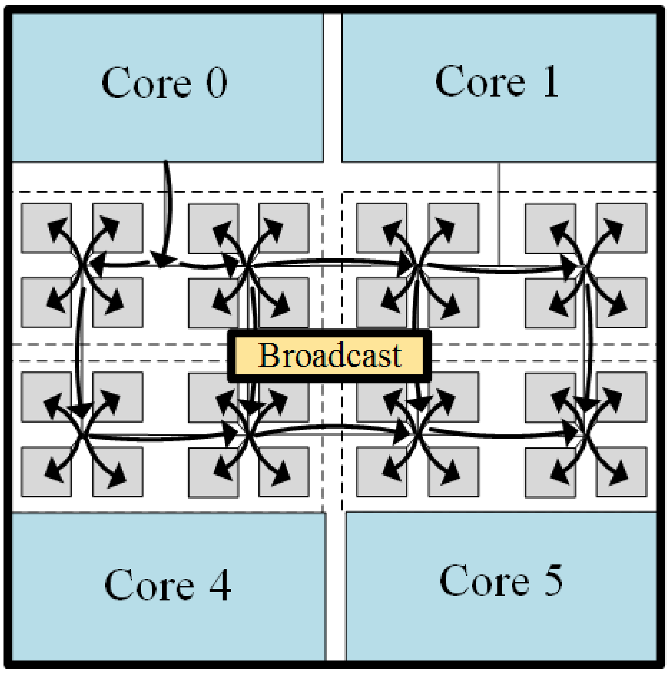


Figure 6.4: Example to illustrate the complexity of data look in Dynamic NUCA Organization

Locating data blocks with no limitation on their possible location, requires a broadcast to all the banks for each access. That would be prohibitive in terms of both latency and energy. Therefore, placement is strongly paired with the lookup mechanism and the greatest challenge is developing hybrid solutions that lay somewhere between the static and the extremely dynamic policies, which would deliver high performance at an affordable cost. Hence, the full potential of the NUCA access latencies are not exploited.

6.3 Proposed Shared Cache Management using AMR

This section presents the details of our baseline architecture to facilitate the explanation of the proposed scheme.

6.3.1 Baseline Architecture

The block diagram of the baseline architecture for L2 cache is shown in Figure. 6.5. As seen from the figure, we have eight cores C_1 to C_8 on the same chip with individual private L1 caches and a large shared L2 cache.

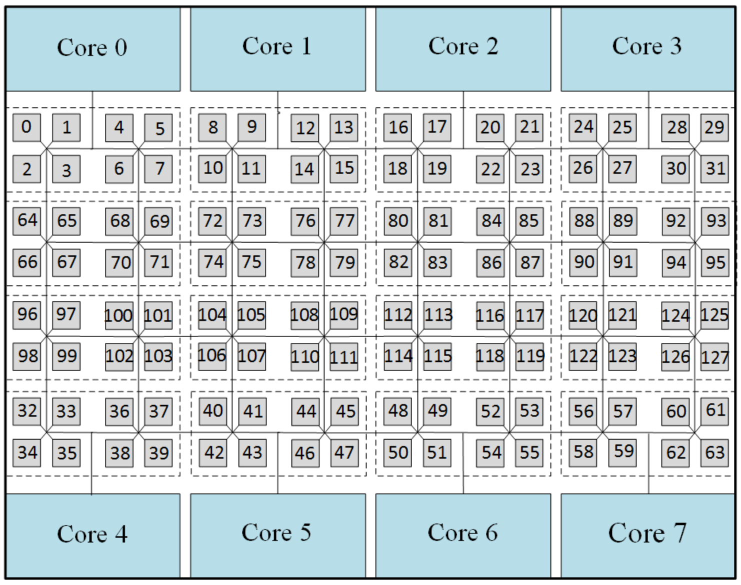


Figure 6.5: Multi-banked NUCA Organization (with Bank ID's indicated)

The L2 cache is further partitioned into multiple banks. We assume a last level shared L2 cache as a Non-Uniform Cache Architecture, derived from Kim et al.'s Dynamic NUCA-1 (D-NUCA) design [16]. We first define few terms to facilitate describing our baseline architecture.

Owner Bank: The bank to which data is mapped for the first time after an off-chip access using a static address mapping scheme.

Bankclusters: A group of eight banks compose a bankcluster and the entire NUCA cache (128 banks) is divided into 16 bankclusters shown by the highlighted portions in Figure 6.6.

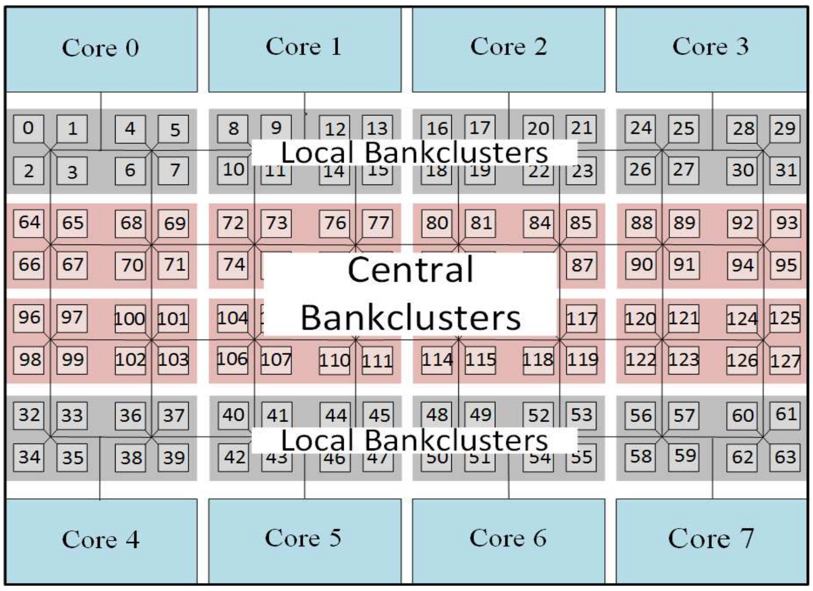


Figure 6.6: Logical Partitioning into Bankclusters

Bankset: All the banks that compose the NUCA cache are treated as a set-associative structure as shown in Figure 6.7 where in each bank holds one way of a logical bankset. Each bankcluster consists of a single bank of a bankset. The mapping of addresses to banks in the local bankcluster and the central bankcluster is presented in Algorithm-1 and Algorithm-2 respectively.

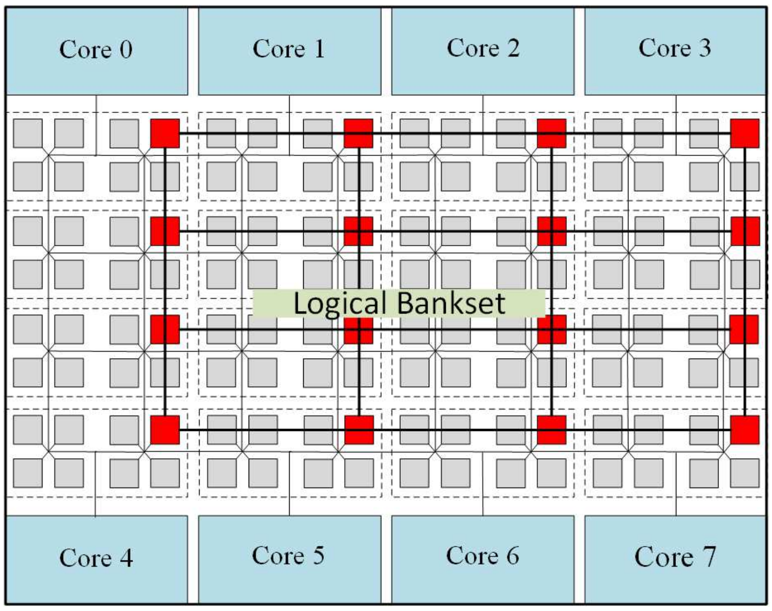


Figure 6.7: Bankset shown in red (16 way bankset associative)

As shown in Figure. 6.7, the entire NUCA cache is partitioned into 128 banks, which is logically organized into a 16-way bankset associative structure (Red colored banks constitute a bankset). Now, the group of eight banks (bankcluster) that are located close to the cores are called local banks (grey colored region in Figure 6.6), whereas the other eight banks that are located at the center of the NUCA cache are called central banks (indicated by light red regions in Figure 6.6). Therefore, in a bankset associative NUCA cache a data block can have 16 possible placements (eight local banks and eight central banks).

Algorithm-1: L1 request mapping to local L2 bank:

```
function mapL1_request_LocalL2_dest
INPUT: L1 ID, num_banksets, CPU Address (addr),
OUPTUT: L2bank $ID_{dest}^{local}$ ,
BEGIN:
        L2bank $ID_{dest}$ = (L1  ID * num_banksets) + addr [$\log_2$ (num_banksets) -1: 0]
END
```

Algorithm-2: L1 request mapping to central L2 bank:

```
function mapL1_request_CentralL2_dest
INPUT: L1 ID, num_banksets, CPU Address (addr), num_L2banks.
OUPTUT: L2bank $ID_{dest}^{central}$ ,
BEGIN:
        L2bank $ID_{dest}$ = (L1 ID * num_banksets) + addr [$\log_2$ (num_banksets) -1: 0] +
(num_L2banks)/2;
END
```

The address mapping of an incoming data block to an L2 bank during its first reference from off-chip memory is statically determined using the lower bits of the data block address as shown in Figure 6.8. The LRU data block in the referenced set of this bank would be evicted if the set is completely occupied by data blocks. Once the data block is placed in a bank of the D-NUCA cache, the migration policy is used to determine its optimal position. Researchers in the past have proposed gradual promotion ('one-hit-one-hop') for data blocks [105] [58]. In Ideal D-NUCA, a data block can be mapped into any cache bank to maximize placement flexibility for the block. However, the overhead of searching a data block in that scenario may be too large as each bank in entire NUCA cache must be searched for the block.

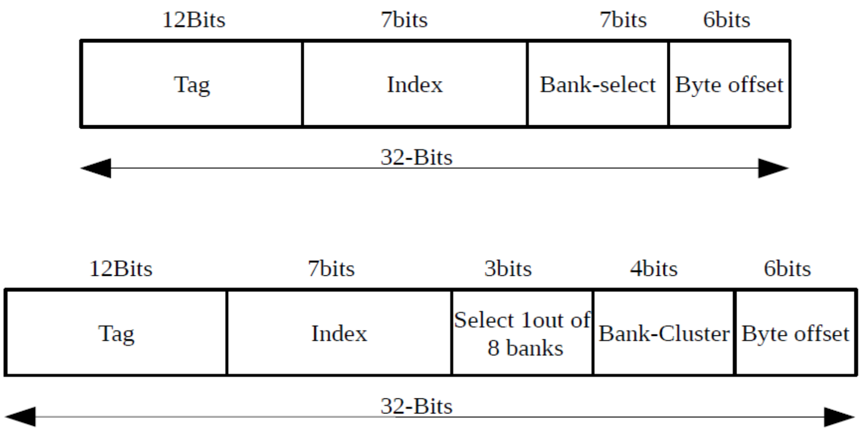


Figure 6.8: Address Interpretation

Previously, data lookup was performed either using centralized tags or by broadcasting the tags to all the banks. Such a policy came at the cost of increased network traffic and higher power dissipation. To address this issue, researchers suggested that data blocks be allowed to be mapped to only one bankset [16] [58]. Such a D-NUCA design uses a two-step multicast data access algorithm. In the first step, it broadcasts a data block request to the local bank that is closest to the core that has initiated the memory request, and to the eight other central banks in the bankset. If all nine requests result in a miss, then in the second step, the request is sent in parallel to the remaining seven banks (central banks) of the requested data's bank-set. Finally, if the request misses in the remaining banks as well, then it is forwarded to the off-chip memory. Therefore, when we evaluate NUCA further, we will assume the same D-NUCA architecture described above in this section, but we will use our proposed data access algorithm (see sub-section 2.3) to find the exact location of data instead of the two step multicast data access algorithm. The traditional D-NUCA access policy is described in Algorithm 3.

Algorithm-3: Baseline D-NUCA data access policy

```
function handleCoreRequest
INPUT: Read/Write request for $Line_j$ ($Req_j$) from $C_i \in C$
BEGIN
1.    Lookup $L1_i$
2.     if (hit)
3.    Load $Line_j$ , $LRUQueue_{set}$ .movetoEnd($Line_j$)
4.     else
5.      $\forall$ k   $BC_{central}$, Fwd ReadReq j $\rightarrow BC_{local}^{(i)}$, k
6.    if (hit)
7.    Load $Line_j$, $LRUQueue_{set}$ .movetoEnd($Line_j$)
8.    else
9.      $\forall$ k   $\{BC\} - \{BC_{central}, BC_{local}^{(i)}\}$
10.   Fwd ReadReq j $\rightarrow$ k
11.   if (hit)
12.   Load $Line_j$, $LRUQueue_{set}$ .movetoEnd($Line_j$)
13.   else
14.   Fwd ReadReq j $\rightarrow$ off-chip
15.   endif
16.   endif
17.   endif
END
```

6.3.2 Working of the Adaptive Migration-Replication scheme (AMR)

When a block is first brought on-chip as a result of a cold miss, it is placed in a bank statically determined by the lower bits of the physical address sent out by the requestor. In cases where it is frequently accessed by a core that is located far away from this bank, this position is far from optimum. A preferred location would be the local bank-cluster of the requesting core. We propose two mechanisms that work in tandem to determine the optimum location for a block on-chip, (i) a gradual mechanism in which the block migrates in steps towards a remote requester, (ii) an abrupt mechanism in which the block is replicated directly to the local bank of the requester. Both these mechanisms require the use of hardware counters to monitor access patterns over a pre-determined time window. Consider the example in Figure 6.9, in which there is a remote hit for a block located in the local bank-cluster of Core 7, but frequently utilized by Core 0. There exists a need to move the candidate block closer to the requesting core (Core 0) in order to reduce hit latency for Core 0. This is achieved by means of cache line reuse tracking using hardware counters.

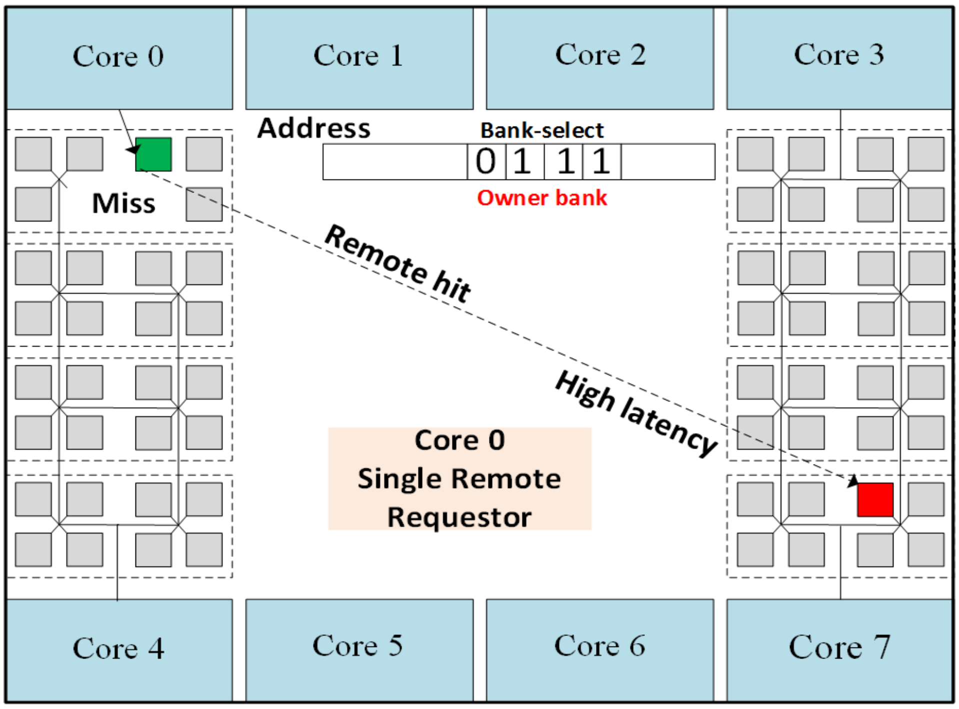


Figure 6.9: Remote hit in the local bank-cluster of Core-7

A 2-bit saturating hardware counter MC (**M**igration **C**ounter) is used with each block to keep track of access patterns from different cores. The core specific migration counter is incremented on every hit from that particular core. Now, if MC saturates after a certain number of accesses (≥ **M**igration **T**hreshold (MT)), the second counter RC (**R**eplication **C**ounter) begins to start incrementing, with MC reset to 0. On every MC saturation, the block is migrated one step closer to the requesting core. The role of this second counter is to decide, whether to provide a separate copy of the requested block at the local bank cluster of a frequent requestor. Therefore, when RC saturates, a replica of the block is placed in the local bank cluster of the requesting core, with the RIB of the replicated block (**R**eplication **I**ndicator **B**it =1) set to 1. Both MC and RC are now reset. Another scenario in which the counters are reset is when the block reaches the local bank-cluster of a frequent requestor after a series of migrations. Figure 6.10 shows the organization of banks, sets and cache lines in the shared NUCA LLC with our novel contributions highlighted.

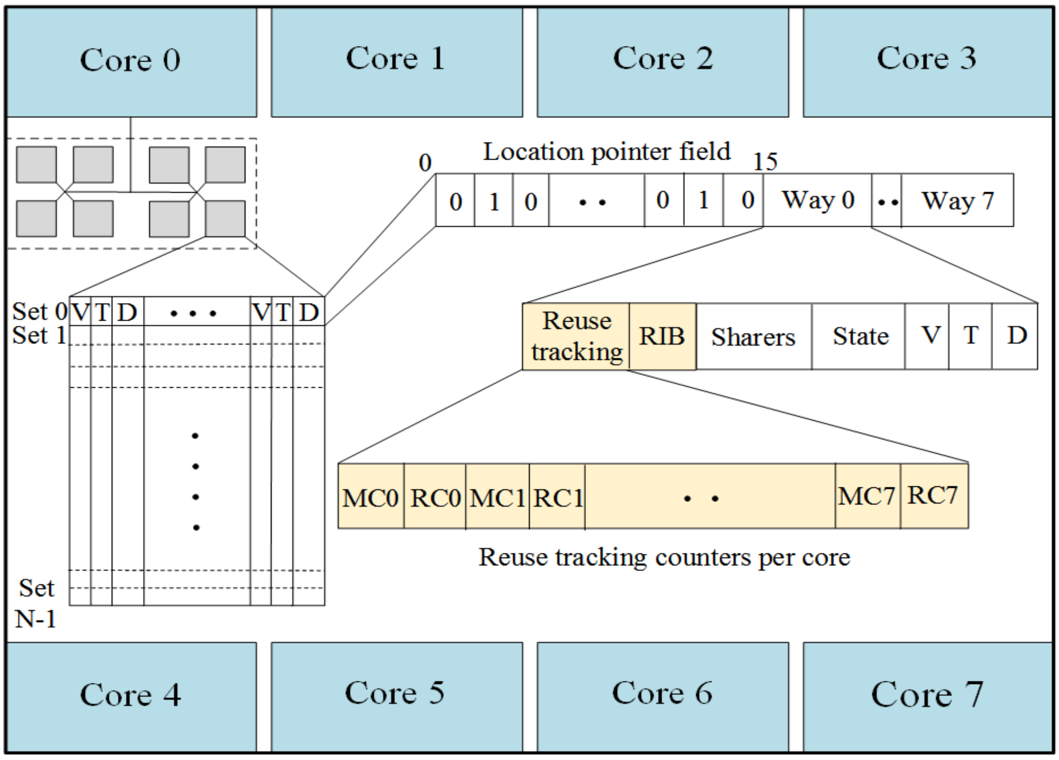


Figure 6.10: Dynamic profiling of block usage with inline directory counters (V: Valid bit, T: Tag bits, D: Data bits)

The following sub-sections explain in detail, the series of steps taken by the proposed AMR scheme in different scenarios.

6.3.2.1 Single remote requestor

The data access pattern of the application suggests a single remote requester. As a result, the block is moved as per our migration policy (from its statically determined location) within the same bankset. The core specific MC increases with each request from that particular core along with block migration towards the requestor when MC saturates. In case, there is no other requesting core then the block will be migrated to the local bank-cluster of the frequent requestor as shown in the Figure. 6.11.

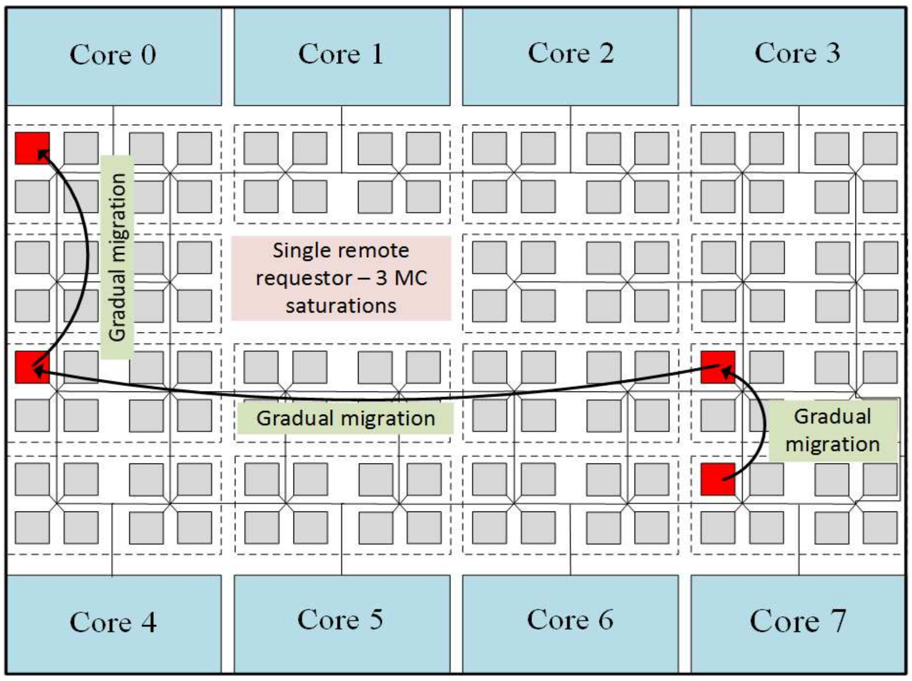


Figure. 6.11. Gradual Block migration

The logic for block migration is presented in Algorithm-4.

Algorithm-4: Block Migration

function handleNUCABlockMigrateRequest

```
INPUT: NUCA cache hit for $Line_j$ ($Req_j$) from $C_i \in C$ , $MC_i^{(j)}$, $Location_j^{(old)}$
OUTPUT: $Location_j^{(new)}$
BEGIN
1. if ($MC_i^{(j)} \geq MT$)
2.     if ($Location_j^{(old)} == Local\ bank_{other}$)
3.         $Location_j^{(new)} \leftarrow Central\ bank_{other}$
4.         $RC_i^{(j)}$ ++
5.     else if ($Location_j^{(old)} == Central\ bank_{other}$)
6.         $Location_j^{(new)} \leftarrow Central\ bank_i$
7.         $RC_i^{(j)}$ ++
8.     else if ($Location_j^{(old)} == Central\ bank_i$)
9.         $Location_j^{(new)} \leftarrow Local\ bank_i$
10.        $RC_i^{(j)} \leftarrow 0$
11.        $MC_i^{(j)} \leftarrow 0$
12.    updateBlockLocation ($Line_j$)
13. else
14.        $MC_i^{(j)}$ ++
15.    endif
16. endif
END
```

6.3.2.2 Multiple frequent requestors

The access patterns of the application may indicate frequent usage of a cache line from multiple cores. In this case, the gradual migration mechanism proposed in [19] would lead to the block 'ping-ponging' between the two competing cores as depicted in Figure 6.12. The block's position may alternate between the two central banks of the requesters or between their local and central banks. With the block's position dynamically varying with each competing request, the block will not be able to eventually migrate to the local bankclusters of either of the cores as discussed in the previous sub-section. In this case, finding out an optimum placement for a data block within the NUCA is a key challenge to avoid ping-pong within the bankclusters of the same bankset. The side-effects of 'ping-ponging' of data blocks includes extra network traffic and subsequently greater dynamic power consumption. To solve this problem, we propose to selectively replicate blocks when 'ping-ponging' of requests is detected between two cores.

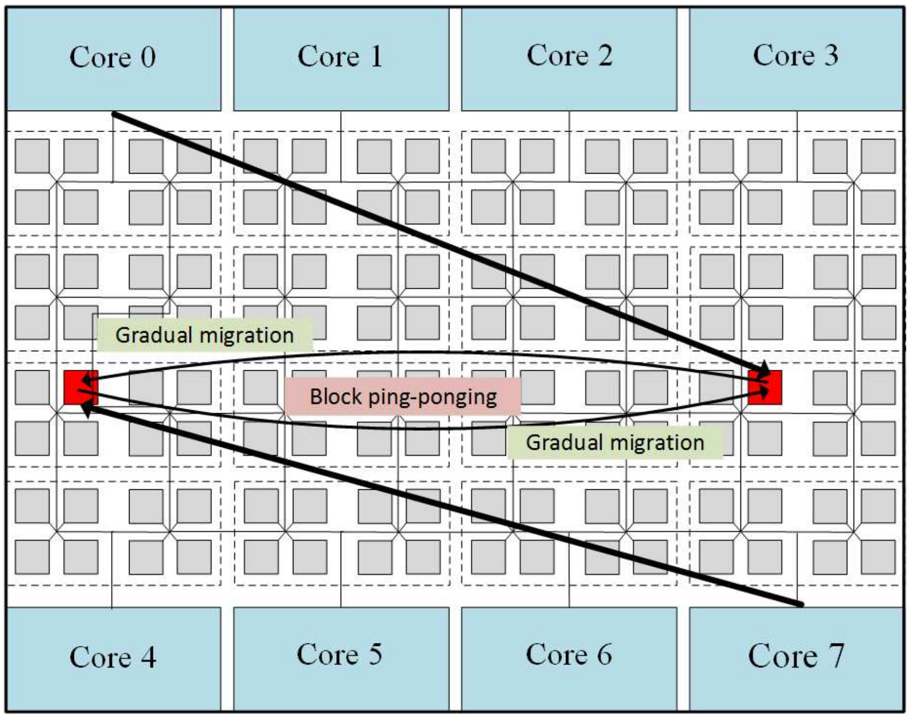


Figure 6.12: Block ping-pong scenario with two competing cores

In the proposed scheme, the cache controller uses the values of both the migration (MC) and replication counters (RC). In case, the value of RC saturates for one of the requesting cores, then the controller creates a copy of the requesting block into its local bankcluster (within in same bankset) and sets RIB to 1 for this block (refer Figure 6.13). The migration and replication counters for both copies of the blocks are reset to 0. The same is true for other frequent requestors as well. Now all the future requests by the competing cores to the same block can be handled at their respective local bankclusters, while requests from the other cores move the original copy of block closer to them, as dictated by the migration policy. As a result of replication, all competing cores that show enough promise (greater reuse as reflected in the saturation of RC) would be given low-latency access to the block, while other cores whose RC for that block is yet to saturate utilize the gradual migration scheme to eventually obtain the block in their local bankclusters. The conditions for block replication are presented in Algorithm 4.

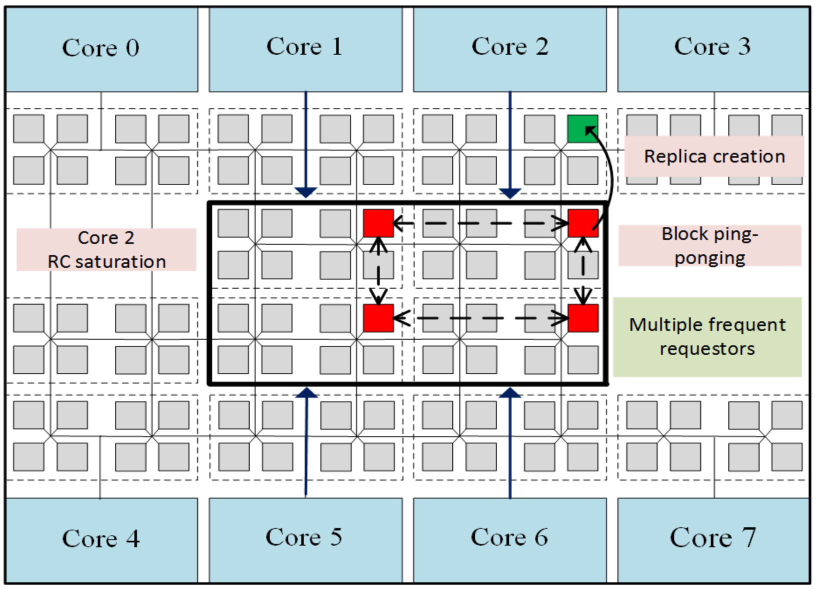


Figure 6.13: Replica creation in the local bank cluster of frequent requestor (Core 2)

Algorithm-5: Block Replication

```
function handleNUCABlockReplicateRequest
INPUT: NUCA cache hit for $Line_j$ ($Req_j$) from $C_i$   $C$ , $MC^{(j)}$, $RC^{(j)}$ , $Location_j^{(old)}$
OUTPUT: $Location_j^{(new)}$
BEGIN
1.      if ($RC_i^{(j)} \geq RT$)
2.      $Location_j^{(new)} \leftarrow$ Local bank $_i$
3.      $RC_i^{(j)} \leftarrow 0$, $MC_i^{(j)} \leftarrow 0$
4.      $RIB^{(j)} \leftarrow 1$
5.      updateBlockLocation ($Line_j$)
6.      endif

END
```

6.4 Proposed Data Access Policy for Shared Last Level Cache

We have seen that D-NUCA uses a migration policy to move data blocks close to the requesting core. This provides low-latency access in an architecture where wire delays significantly impact processor performance. However, such a dynamic data movement within NUCA banks comes at the cost of a complex data access policy. Designing an efficient and low-cost data access policy is very challenging. In order to simplify the complexity of an ideal D-NUCA, we restrict data movement within a group of banks called a bankset. Now, in order to keep track of the location of the *migrated* and *replicated* blocks on-chip, we extend each set within a bank with a p bit location pointer (for the p banks) as shown in Figure. 6.14.

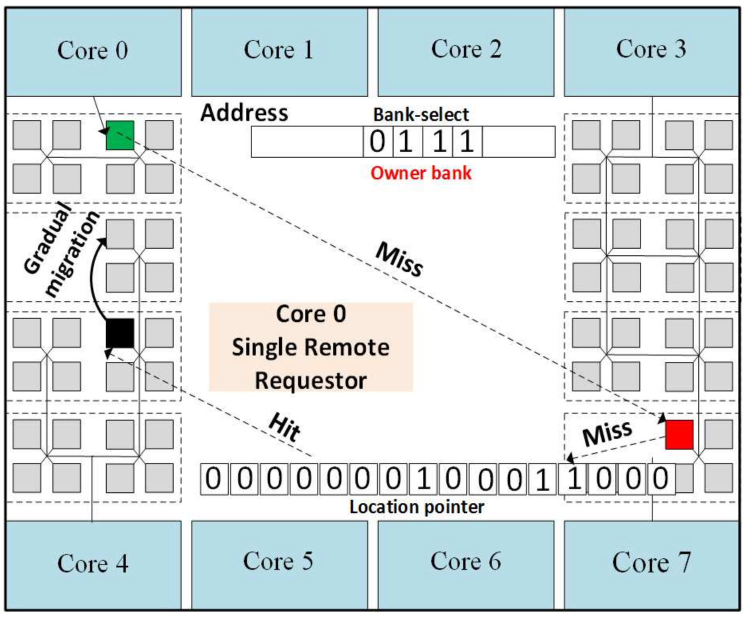


Figure 6.14: Location pointer co-located with each set

Each bit (denoted by the 1's in the location pointer field) indicates the possibility that the cache block is located in that bank, either due to migration or replication. There is a separate RIB with each cache line that indicates whether the cache line possesses a replica in another bank.

Further each bank acts as an owner bank for an equal number of blocks on the chip. This assignment is done statically using the lower bits of the requestor's physical address. This static assignment ensures that every bank is given a fair chance to be the owner of an equal number of cache lines and helps in load balancing at the owner bank as all on-chip block requests that miss in the local bankcluster are serialized at the owner bank. Based on the bits which are set in the location pointer field, requests are sent to different banks. The number of responses received vary based on the following cases:

1. If the block has no replica and has migrated away from the owner bank, a single response is received.
2. If the block has no replica and is located at the owner bank, then the owner bank services the request for the block.
3. If the block has a copy in another bank, responses from two or more banks may pollute the on-chip network.

We handle the last case separately to ensure that the requestor is serviced from a single bank that contains the block. The detailed logic is presented as follows. On a local bankcluster miss, the owner bank location pointers are probed to identify possible locations of the cache block. Requests are sent to all banks whose bit is set in the location pointer field. We ensure a single responder by adding additional circuitry at the tag comparison stage. We propose that only the original copy of the block (RIB=0) must service the new request in order to avoid additional coherence complexities in the presence of multiple responders. Our additional circuitry does not increase the latency of the tag comparison and can be done in parallel with the check that is performed to ensure that the block contains valid data.

The location pointers need to be updated in case a block has migrated or a replica has been created. Section 6.5 explains the mechanism to update the location pointers in detail. Previous migration based approaches have either used broadcast or partitioned multicast as their search policy. In contrast, using the location pointers in the owner bank, we can efficiently direct our search to a subset of banks at the cost of a very low hardware overhead (6.8% including reuse tracking).

Algorithm-6: Block search

```
function searchCacheBlock

INPUT:  NUCA cache request for $Line_j$ ($Req_j$) from $C_i$   C
BEGIN:
1.      Fwd $Req_j \rightarrow$ Owner Bank $^{(j)}$
2.       if (hit)
3.      $L1_i \leftarrow$ Load $Line_j$
4.      else
5.          $\forall$ k   {Bits set in location field}
6.    Fwd $Req_j \rightarrow BankInBankset_k$
```

```
7.      if (hit && $RIB^{(j)}$ !=1)
8.      $L1_i$ ← Load Line $_j$
9.      else
10.     Fwd Req $_j$ → off-chip
11.     Owner Bank $^{(j)}$ ← Address [Bank-select bits]
12.     $L1_i$ ← Load Line $_j$
13.     endif
14.     endif
END
```

Algorithm-7: Updating Location Pointers

function updateBlockLocation

INPUT: $CacheLine_j$, updateCause

BEGIN

```
1.      k ← findOwnerBank($CacheLine_j$) //Static mapping
2.     if (updateCause == MIGRATE || updateCause == REPLICATE)
3.      if (numBlocksInSetWithOwner $_{dest}$ (k) == 0 )
4.      LocationPtr $_k$ [dest]  ← 1
5.      if (numBlocksInSetWithOwner $_{src}$ (k) == 0 )
6.      LocationPtr $_k$ [src]  ← 0
7.      endif
8.      endif
9.      else if (updateCause == EVICTION)
10.     if (numBlocksInSetWithOwner $_{src}$ (k) == 0 )
11.     LocationPtr $_k$ [src]  ← 0
12.     endif
13.    endif
END
```

6.5 Updating location pointers

To begin with, the location pointer bits and the RIB are reset to zero (invalid cache block). The ‘p’ bit set pointer and the RIB are updated in the following scenarios.

1. When the block is first brought into the owner bank from off-chip memory, the location pointer field corresponding to the owner bank is set (if it has not already been updated by another block belonging to the same set and owned by the same bank). The RIB field for the block is reset.
2. When the block migrates on a remote hit, the owner bank is made aware of the destination bank for that block, and the bit corresponding to the destination bank is set in the location pointer field of the owner bank.
3. When a ping-pong is detected and a replica is created, the RIB corresponding to the replica block is set and the bank locations of the replica are updated in the location pointer field of the owner bank for that block.
4. When a block (not having a replica in the same bankset) is evicted to off-chip memory, the set is examined to see if it has any other blocks with the same owner bank as the evicted block. If yes, then the location pointer field in the owner bank is left unchanged. If not, the bit corresponding to the evicted block’s bank in the location pointer field of the owner bank is reset to 0.
5. When a replica is invalidated, either due to an exclusive write request or write-back request or when it has shown less reuse (LRU), the current bank holding the replica is examined to see if it holds other blocks with the same owner bank as the replica. If yes, then the location pointer bit field is left unchanged. Otherwise, the owner bank is notified to reset the location pointer field bit corresponding to the bank from which the replica was evicted.

6.6 Coherence Protocol

Researchers have been extensively working on managing on-chip coherence for shared caches in CMPs. Different cache coherence protocols have been proposed to keep data coherent in a multicore environment. This section presents the working of cache coherence protocols as adapted to our proposed scheme. It is based on the basic MESI protocol to maintain cache coherence and correctness. Figure 6.15 shows the additional bits required to maintain the list of sharers and coherence state at the L1 cache and L2 cache.

6.6.1 False miss

In non-uniform shared last level caches that allow for migration, an important issue to be addressed is the handling of requests to blocks in transit during the migration process. As the request misses in both the source and destination banks, the requestor wrongly infers the absence of the block on-chip. This problem has been referred to as the 'false miss' problem in literature and can lead to costly off-chip misses (refer Figure 6.16). With two copies of the block now present on-chip, if either of the copies is modified, it becomes impossible to maintain coherence between the blocks.

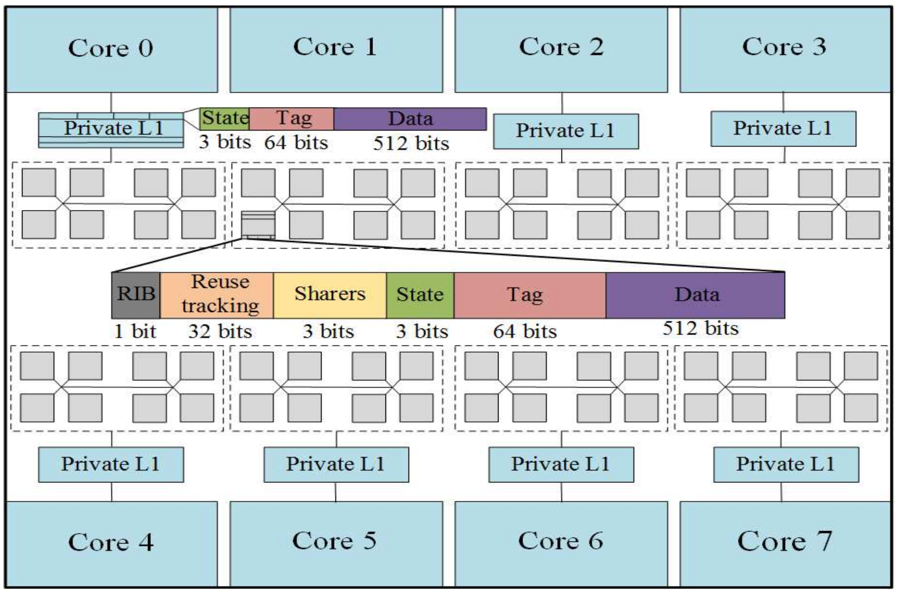


Figure 6.15: Additional bits within cache line to maintain coherence and reuse tracking

To solve this problem, we use a two-way handshake between the source and destination banks. On a remote hit, the source bank sends both the cache line and a 'Migration:Begin' message to the owner bank. The destination bank on receiving the data block responds with a 'Migration:End' message to the owner bank. The owner bank now acknowledges both the source and destination banks after updating the location pointers, with 'Migration:Ack; messages. Now the source de-allocates the cache line. Requests received by the source during the transition are serviced at the source and requests received after the 'Migration:End' message are forwarded to the destination by the owner bank. We will explain the working of the proposed coherence protocol and the mechanism to handle false misses through the following access scenarios.

6.6.2 Read request

6.6.2.1 Hit in the local bankcluster:

If the GETS (shared read) request hits in the local bankcluster of the requesting core, then it is directly transferred to the private L1 cache of the requestor.

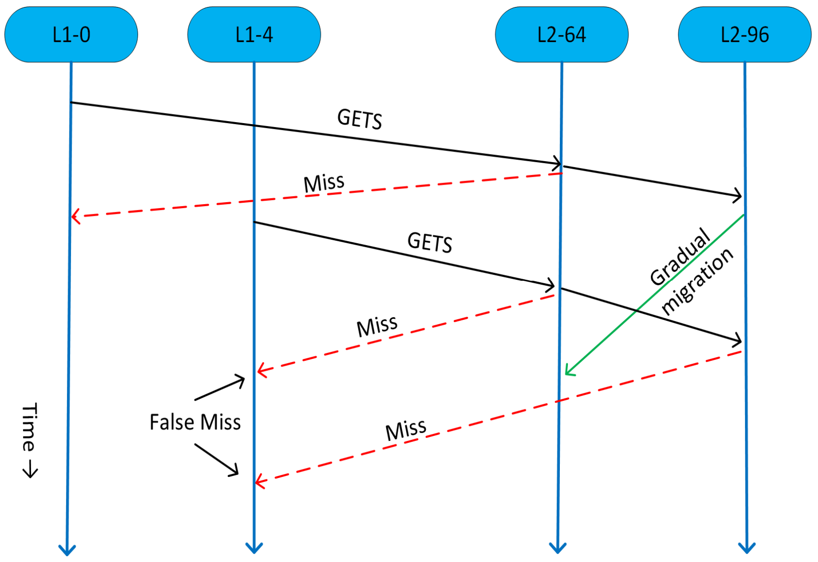


Figure 6.16: Core-4 facing a false miss due to block migration

6.6.2.2 Miss in the local bankcluster:

6.6.2.2.1. Replica absent: On a miss in the local bank-cluster of a requesting core, the request is forwarded to the owner bank. If the block is present in the owner bank, it is sent to the L1 cache of the requestor, otherwise the location pointers are examined and the request is selectively broadcasted to all the banks whose bits were set in the location pointer field. There exists a need to selectively broadcast search requests as locations are tracked at each set and a single set may contain other blocks with the same owner bank as the requested block, although the requested block may itself not be present in a particular bank. Now, the bank that contains the requested block (tag match) responds to the requesting core and the block is transferred to the L1 cache of the requestor. Further, this block migrates from the initial bank toward a bank closer to the requesting core, if MC saturates, otherwise it will remain in the same bank (MC incremented). Figure 6.17 explains the migration process in detail. Consider a GETS request from Core 0's L1 cache that misses in the local bankcluster. Using the information provided by the location pointers at the owner bank the block is found to be located at L2-36. On a hit in the remote bank (L2-36), MC4 saturates and the block migrates one hop closer to Core 0. After three such MC4 saturations, the block migrates three hops (L2-36 → L2-100 → L2-68 → L2-4) and is placed in the local bankcluster of Core 0.

Requests during the transit are handled as described in the previous section to avoid false misses. The detailed sequence diagram is presented in Figure 6.17.

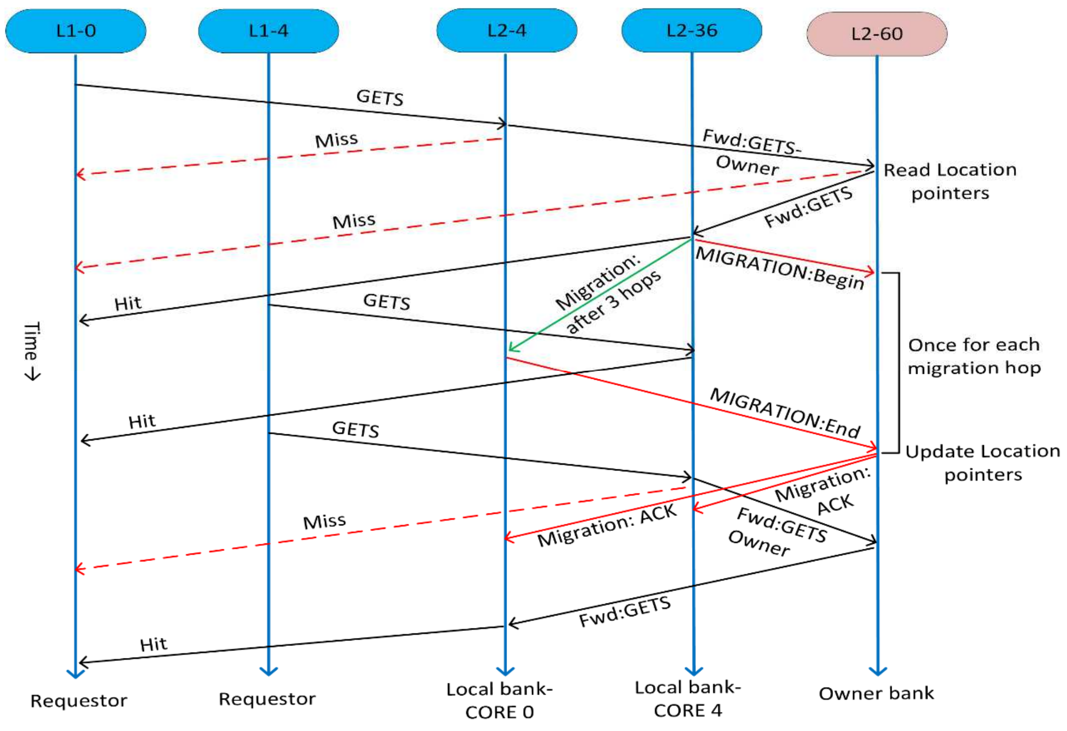


Figure 6.17: Migration mechanism to handle read requests (replica absent)

6.6.2.2.2. Replica present in the local bankcluster of requesting core: In this case, the shared read request (GETS) can be handled directly at the local bankcluster that stores the replica.

6.6.2.2.3. Replica(s) present in the local bankcluster of other cores: In this case the request is forwarded to the owner bank. If the data is present in the owner bank then the data is transferred to the L1 cache of the requesting core. In case there is a miss at the owner bank then the location pointers are examined and subsequently the request is sent in parallel to all the banks in the bankset, whose location pointer field is set to 1. In this case, however, multiple banks respond to the request. In an attempt to reduce on-chip traffic created by allowing multiple responders to send their data blocks, we choose only that data block, for which RIB is not set. In contrast, an approach that receives data from all the responders and then chooses to ignore the later received blocks would significantly limit on-chip bandwidth for other requests. By making this optimization we also reduce coherence protocol complexity and save energy. The requester on receiving the data block, acknowledges with a 'DATA_ACK' message.

6.6.3 Exclusive write request

6.6.3.1 Replica absent in bankset: If the write request hits in the local bank-cluster or in a remote bank-cluster as determined by the location pointer bits of the owner bank, all other sharers of the line are sent invalidation requests (INV). On receiving acknowledgements from the sharers (INV_ACK), the requestor is given exclusive rights to the line. The block is transferred to the L1 cache of the requestor and written (refer Figure 6.18)

6.6.3.2 Replica exists in the same bankset: GETX Request

Multiple copies of the cache line are present in the bank-set and a single core issues GETX request. There are three different cases as per our lookup policy.

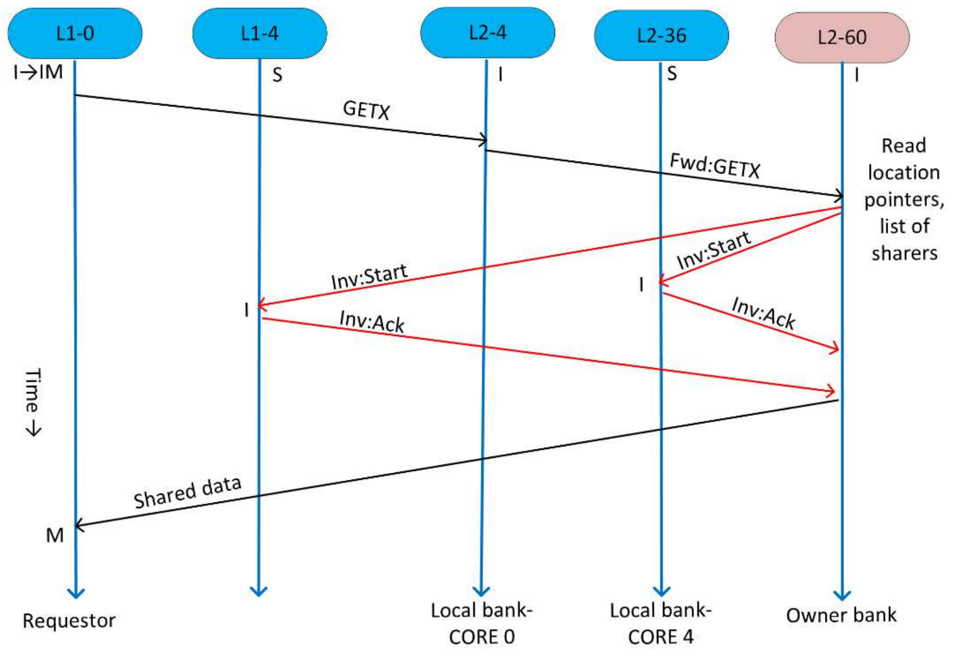


Figure 6.18: Sequence diagram showing the invalidation steps in case of write requests

1. Replica exists in the local bank-cluster of the requesting core (Exclusive or Modified State): On a L1 cache miss for a GETX request from a core, the request is forwarded to the local bankcluster of the same core. If a replica is present in the E or M state, then the cache block can be directly transferred to the private L1 cache of the requesting core.

2. Replica exists in the local bank-cluster of the requesting core (Shared state) or Replica exists in the local bankcluster of other cores (S/E/M state): In this case, the request is forwarded to the owner bank of the block (where the list of sharers of the block are maintained). The owner bank sends invalidation message to both L1 and LLC copies of the block. The location of the replicas is determined at the owner bank using the location

pointers. Once the invalidation acknowledgements (INV_ACK) are received, the requesting core can be granted exclusive access (E) to the line in its private L1 cache. The invalidations mentioned above are essential to maintain the 'single-writer multiple reader' invariant necessary for the correct operation of coherence protocols.

6.6.4 L1 evictions

Consider the case when an incoming block evicts a dirty L1 line in a write-back cache.

6.6.4.1 No replica in the bankset: The L1 cache controller issues a PUTX (write-back) for that line. The request is sent to the owner L2 bank and the set pointers are examined to find the location of the line. The PUTX requests are selectively broadcasted to the banks whose bits are set in the location pointer field. Once the line is found the dirty data block is written into the corresponding L2 bank (refer Figure 6.19).

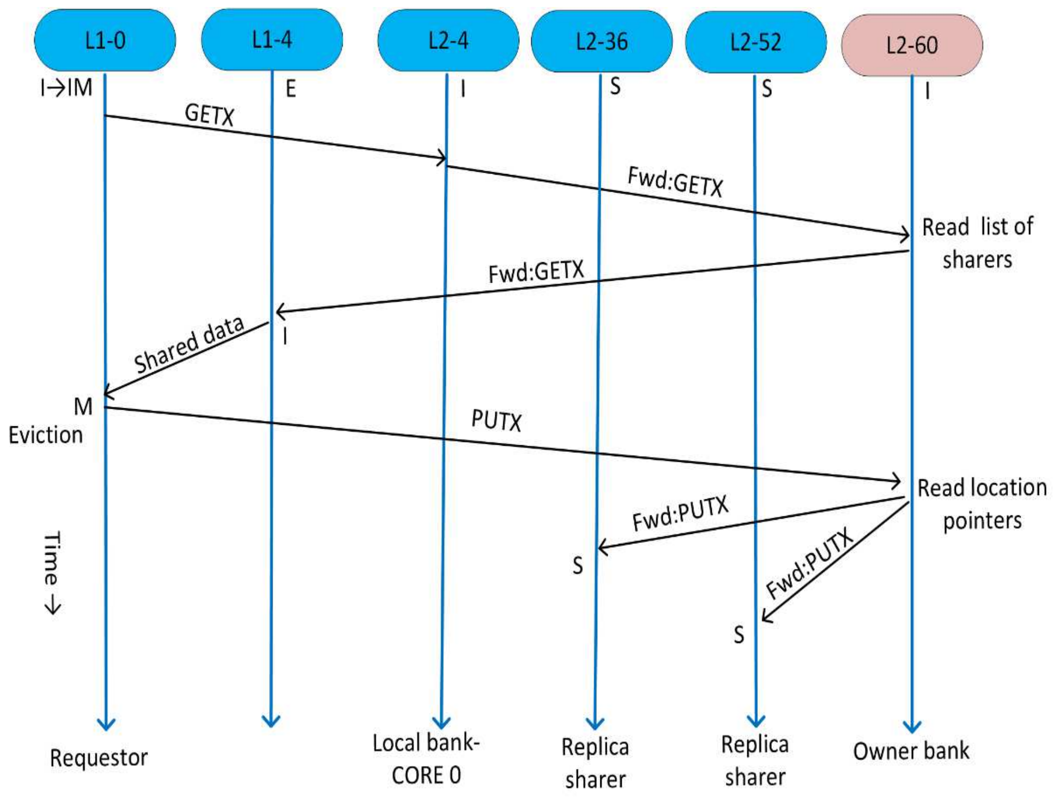


Figure 6.19: Handling L1 evictions (no replica in bankset)

6.6.4.2 Replica(s) present in the bankset: In this case, the request is forwarded to the owner bank, and using the information obtained from the location pointer field and the replication indicator bit (RIB) the dirty L1 block is merged with the LLC replicas.

6.7 Special cases

1. Two simultaneous PUTX requests for the same cache line issued by two different L1 cache controllers: There is no possibility of a race condition arising in this case as both requests are forwarded to the owner bank and serialized before updating the replicated blocks.
2. When a GETX request is issued by an L1 cache for a block owned by another L1 cache, there exists a possibility that the write-back happens to the wrong L2 bank because of gradual migration on a remote hit. This possibility is ruled out in our design because the location pointers are updated in synchronism with every migration/replication event. PUTX requests read the updated location pointers and can be satisfied at the correct L2 destination bank.

6.8 Evaluation Methodology

We evaluated the proposed AMR scheme (Algorithms) on an 8 core CMP. The basic system configuration parameters used for the evaluation are shown in Table 6.1.

6.8.1 Multicore System:

All the experimental evaluations are performed using a single CMP that consists of eight UltraSPARC IIIi homogeneous cores. The cache hierarchy, on-chip interconnection network and cache coherence protocols are **simulated using** the Virtutech Simics full-system simulator [99] that is extended with the GEMS toolset [100]. GEMS simulator provides Ruby, which is a detailed memory sub-system simulator that provides support to implement the proposed cache hierarchy within our baseline system. Each processor core has its own first-level cache (data and instructions) and is connected to a node of the network. The last level of the memory hierarchy is the D-NUCA baseline distributed in 128 banks and connected to the cores via switches.

Table 6.1: System Configuration

Configuration parameters	
No. of cores	8
Core mode	Single thread
Frequency	1 GHz
L1-Data Cache	32 kB, 64 bytes
L1-Instruction Cache	32 kB, 64 bytes
Shared L2 cache	8 MB, 128 banks
Bank size	64 kB, 8-way, 64 bytes

We used MESI based directory protocol to maintain coherency of the memory subsystem.

6.8.2 Benchmarks

Our full system simulator runs an unmodified Solaris 10 operating system. To analyze the proposed schemes, we run selected multithreaded applications from Princeton PARSEC 2.0 benchmark suite [108]. We also run a set of single-threaded applications from SPEC2006 suite. All the application are first compiled using gcc (provided with the Sun Studio 10 suite). The method for the simulations involves first skipping both the initialization and thread creation phases, and then fast-forwarding while warming up the cache for 500 million cycles. Then a detailed simulation is performed for the next 500 million cycles.

6.8.3 Energy

To estimate the energy consumed by the baseline multi-banked NUCA cache and the off-chip memory, we adopted an energy model given by Bardine et al. [103]. This allowed us to calculate the dynamic energy dissipated by the banks in the LLC cache using the Orion tool. To calculate the energy consumed during an off-chip memory access, we have used the micron datasheet. Therefore, the total energy consumed by the NUCA memory system is the sum of all three components:

$$E_{dynamic} = E_{network} + E_{banks} + E_{off-chip}$$

6.9 Results

We selected few applications from the PARSEC 2.0 (Blackscholes, Bodytrack, Canneal, Streamcluster, Swaptions and Fluidanimate) and the SPEC2006 benchmark suites and simulated their execution on the baseline S-NUCA, D-NUCA configurations as well using the proposed policy. We have compared 3 different LLC management schemes and chosen completion time, energy consumption and network traffic (bytes-per-instruction) as the reference evaluation metrics. We have also analyzed L2 access latency in all the three cases to further evaluate our proposed scheme.

6.9.1 Performance Evaluations

The percentage of improvement is obtained by taking the difference between the average value along all the applications for reference and proposed schemes. Figure 6.20 compares the completion time (normalized) for selected applications. It was observed that the proposed

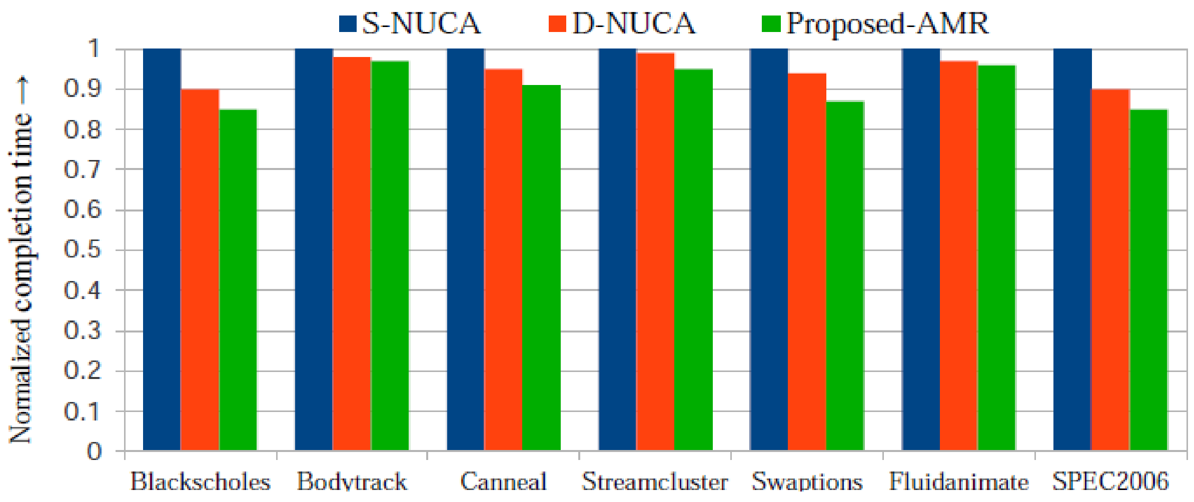


Figure 6.20: Normalized completion Time

On an average we obtain a 9% performance improvement with respect to S-NUCA (baseline) and nearly 4% improvement with D-NUCA.

6.9.2 Network Traffic

A comparative evaluation of the variation in on-chip network traffic is shown in Figure 6.21. It is based on the distribution of both data as well as control messages that affects the overall network traffic (measured in terms of bytes-per-instruction). In our architecture the size of control message is 8 bytes (header only) and the size of the data message is 72 bytes which contains 8 bytes for the header portion and 64 bytes for the data block. It can be observed from Figure 6.21 that the proposed AMR policy reduces the contribution of data messages to the overall network traffic when compared to S-NUCA. The main reason for this reduction is the selective data block replication in the local banks and the migration of data blocks towards the requesting cores, which reduces the number of network hops that must be traversed by a data packet to reach the destination node.

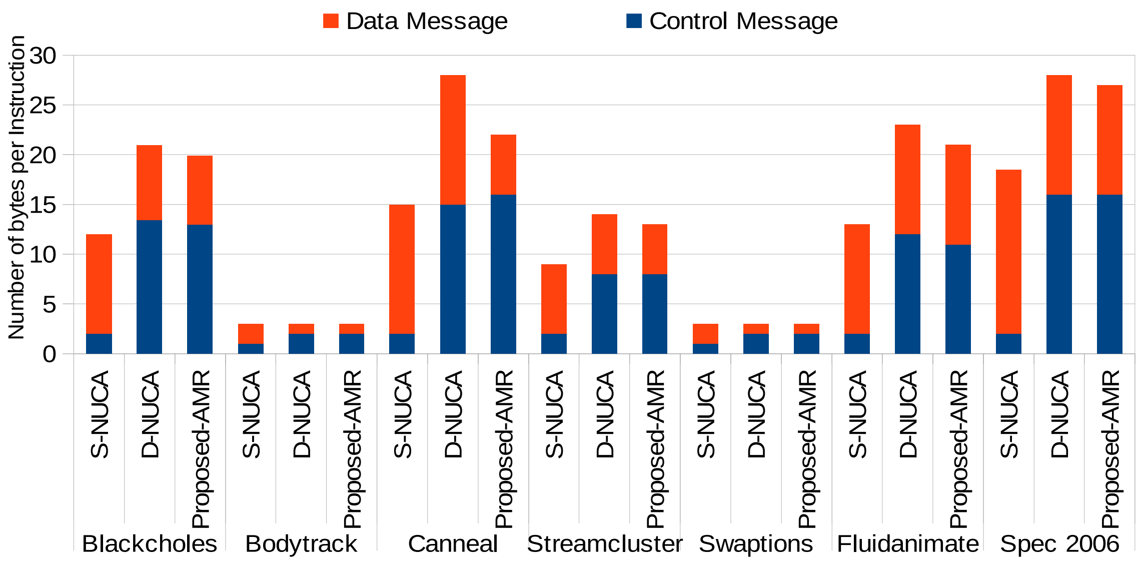


Figure 6.21: Normalized Network Traffic

In AMR, on an average the data packets traverse less number of hops with respect to D-NUCA as well, bringing about a reduction in network traffic. We have also noticed that the control messages are increased for both AMR and DNUCA (conventional, multi-cast search) as compared to the S-NUCA due to the extra messages needed by both of the selective replication and migration schemes to maintain coherency and track data block location. Another important observation that can be made is that the proposed AMR, by virtue of using an efficient data access policy is able to reduce the overall network traffic when compared to both S-NUCA and D-NUCA. The reduction in the network traffic has a direct effect on reducing dynamic energy consumption.

6.9.3 Energy Consumption

Figure 6.22 presents the energy consumption for the three different schemes. We have normalized the results obtained from AMR and D-NUCA with respect to the baseline S-NUCA for each application. Our results include the energy consumed (static and dynamic) by the on-chip network, the last level NUCA cache, and the main memory. As seen from the graph, both D-NUCA and AMR consume lower energy for almost all applications when compared to S-NUCA. As explained in sub-section 6.3.2, by selectively replicating blocks to the local bankcluster of frequent requestors and by broadcasting search requests to only a subset of banks of the bankset, we obtain 5.3 % and 2.3 % energy savings when compared to S-NUCA and D-NUCA respectively.

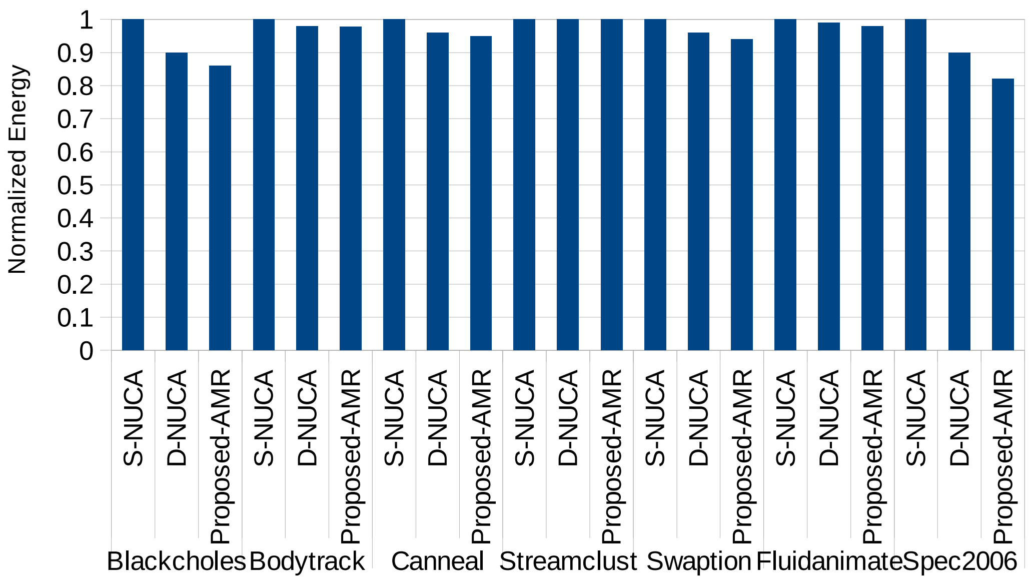


Figure 6.22: Normalized Energy Consumption

To summarize, our simulation results shows that the proposed AMR scheme performs better than both the widely used S-NUCA and D-NUCA LLC management schemes (in terms of completion time and energy consumption). The proposed inclusion of line migration together with the selective replication scheme has considerable utility in improving LLC NUCA performance. We have also observed significant reduction in the network traffic (refer Figure 6.21) and the average L2 hit latency (refer Figure 6.23) when compared to the two state-of-the-art schemes (S-NUCA and D-NUCA).

6.10 Related work

Kim et al. [19] was the first to introduce the non-uniform cache architecture (NUCA) for the last level shared cache as shown in Figure 1. In shared NUCA, the entire LLC is divided into smaller equal sized banks such that nearer cache banks have lower access latencies as compared to farther banks, thus mitigating the effects of increasing on-chip wire-delays.

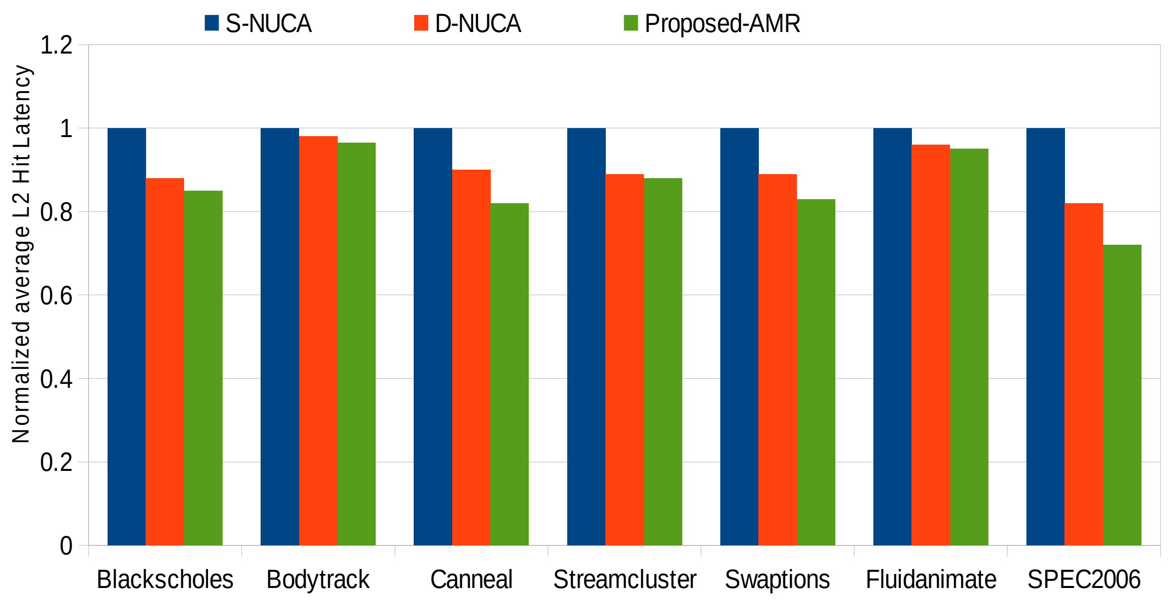


Figure 6.23: Normalized average L2 Hit Latency

The benefits that can be obtained from a cache line migration policy are limited by the effectiveness of a data access scheme that was difficult to implement in the past. Kim [19] was the first to present the importance of the bank access scheme in D-NUCA organizations. Although block migration enhances D-NUCA to outperform S-NUCA, it is limited by the ping-ponging of data between requesting cores and the efficiency of the bank access scheme within NUCA. We address both these issues using selective replication and an efficient search scheme in this chapter. Beckmann and Wood [112] have shown that cache line migration is not beneficial in NUCA cache architecture for multicores as approximately 50% of the hits in commercial and scientific applications are in central bank-clusters to access shared cache lines. In-order to obtain performance benefits they have however used a costly data access policy. Huh et al. [63] proposed a 16 MB dynamic non-uniform cache architecture with 16 cores, but their proposed migration policy ignored the problem of cache line ping-ponging between bankclusters. CMP-NuRAPID [134] uses block replication in NUCA caches, but this policy has ignored last level cache pressure due to block replication. Reactive-NUCA [139] favors instruction replication but has neglected shared block replication in the LLC. Victim replication [116] uses a static policy to replicate blocks that is not effective for all applications, and Adaptive Selective Replication [60] only allows to replicate shared-read-only data and ignores other types of data.

To summarize, for programs that exhibit high degrees of sharing, a majority of the proposed schemes have not been able to combine an efficient block migration scheme with the low latency benefits provided by block replication. Efforts that do so, lack an efficient search scheme that provides fast access to blocks. We have addressed this issue in this chapter by using location pointers that brings about significant reduction in on-chip network traffic and energy consumption at the cost of negligible hardware overhead.

6.11 Summary

We present an adaptive migration-replication scheme (AMR) for shared last level NUCA cache, which dynamically tracks cache line reuse frequency and replicates cache lines that show high reuse to the local bank-cluster of the requesting cores. Our proposed policy determines when and where to migrate cache blocks in tandem with the replication decision. On a set of chosen multi-threaded and single threaded applications, the proposed AMR policy reduces overall energy consumed by 5.3% and 2.3 % and the completion time by 9% and 4% when compared to the S-NUCA and D-NUCA LLC cache management policies respectively. The coherence complexity of our protocol is almost identical to that of a traditional non-hierarchical (flat) coherence protocol since replicas are only allowed to be created at the LLC slice of the requesting core. Our proposed policy is implemented with an extra storage overhead of 6.8% per NUCA bank.

Chapter 7

A Novel work-load aware adaptive Cache

This chapter presents a novel reconfigurable cache architecture to improve cache capacity and reduces on-chip network traffic to improve system performance.

Chapter 7

A Novel Workload-aware adaptive Cache

In this chapter, we have proposed a novel reconfigurable cache architecture to improve cache capacity and reduces on-chip network traffic to improve system performance.

7.1 Introduction

In the previous chapters of this thesis, we have focused on the various cache management challenges in the moderate to large shared L2 caches for CMPs. For that study, we have assumed a cache hierarchy with private L1 caches and shared L2 cache organization with a uniform/non-uniform access latency and physical mapping of blocks to the shared L2 cache. Figure 7.1 presents the memory hierarchy along with trade-off in speed vs size. The efficiency of current high-performance shared memory multicore processors depends on the design of the on cache hierarchy and the coherence protocol. Traditional and current processor cache hierarchies uses a fixed size of cache block in the cache organization and in the design of the coherence protocols.

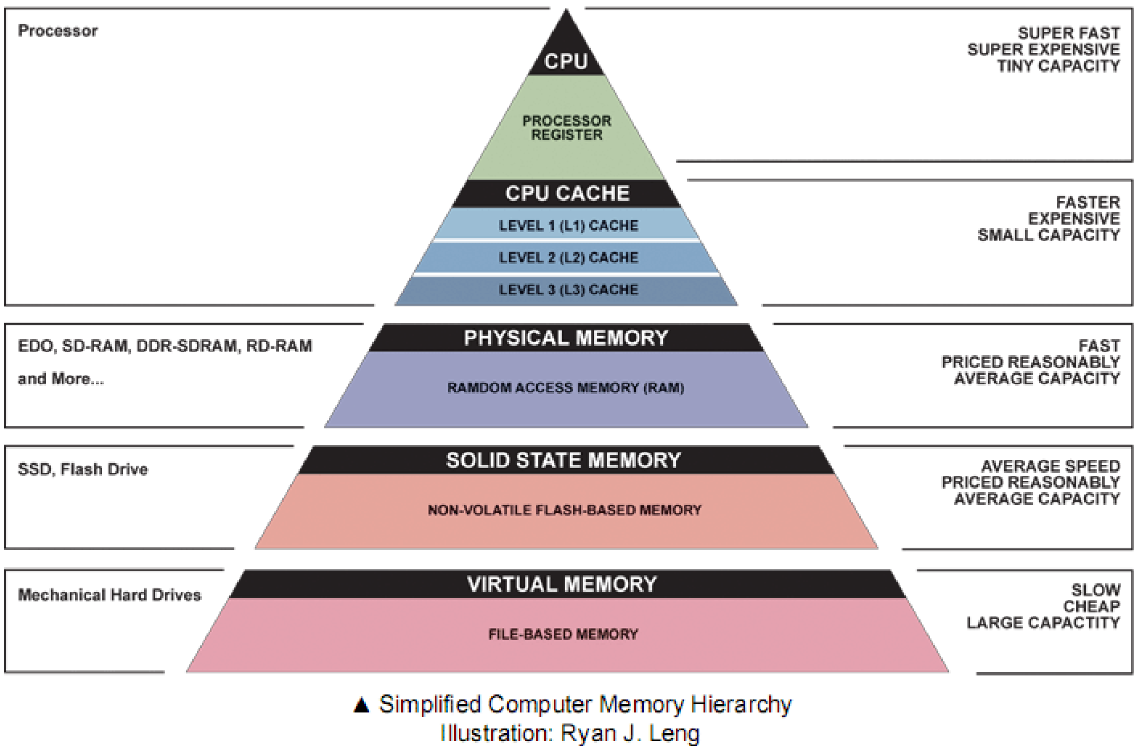


Figure 7.1: Cache Hierarchy and trade-off between size and latency

The fixed size of block in the set is chosen to match average spatial locality requirement across a range of applications, but it also results in wastage of bandwidth because of unnecessary coherence traffic for shared data. The additional bandwidth has a direct impact on the overall energy consumption. In this chapter, we present a new adaptable cache design that can be dynamically reconfigured to match the data movements for an executing applications and its required spatial locality.

7.2 Motivations

Caches memories are designed to exploit locality of reference in order to take benefit of data reuse by speeding up subsequent access to the same data block. There are two different types of reference locality which cache designer try to exploit are temporal and spatial. Present day processors use eight bytes of data at a time and private caches are designed to keep small amount of data that is frequently used near the processor to exploit locality within executing applications. Cache design are either direct mapped or set-associative where each block from memory maps to a single entry in the cache (single way) or in one of the many (number of ways) possible entries in the cache. Figure 7.2 presents a four way set-associative cache structure with data and tag array, where size of each data-block size location is 64bytes. As shown in figure 7.2 each set can store a fixed number of data blocks and that depends on the set-associativity of the cache. Each data block entry (64-bytes) within a set is called a way.

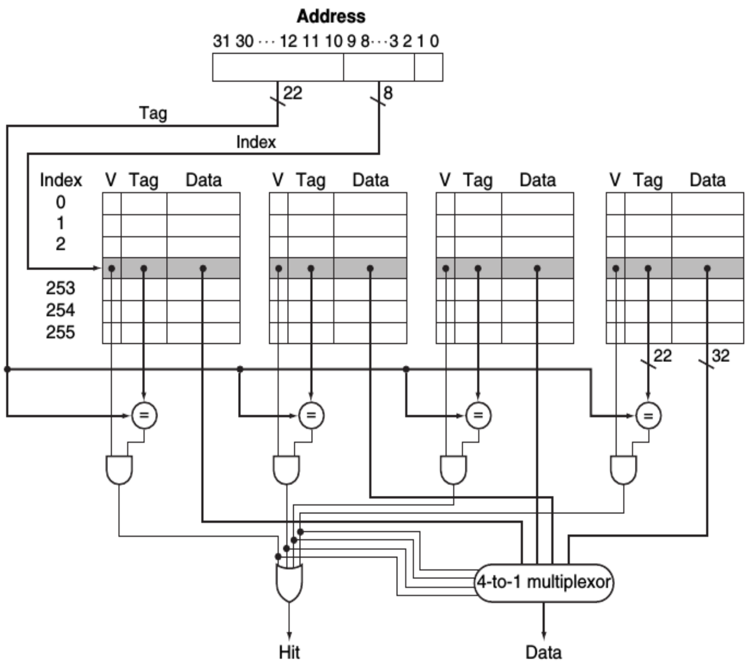


Figure 7.2: Set associative cache with fixed data block size

Therefore, we have observed that the size of the cache block entry is the basic unit of data transferred or allocated in the cache architectures. Data blocks size in caches affects multiple system performance metrics including on-chip interconnection bandwidth, cache miss rate, and cache utilization.

Uniformly sized data blocks simplifies cache requests, and support simple tag organization. However, traditional caches are not flexible because of fixed data block size and fixed number of data blocks in the set which results in poor caching efficiencies for applications that has low spatial locality. We have analyzed that the cache block size exploits spatial locality by naturally prefetching all the neighboring words at the same time.

However, few words in a data block could be evicted untouched during the life cycle of a cache block, due to the varying spatial locality of executing applications. These unused words in the data block consume interconnect bandwidth and pollute the cache, which increases the number of misses. We have analyzed the influence of a fixed size data-block as shown in Figure 7.3 and presented a novel modified cache design with adaptive cache block size depending on the application executed on the processor.

7.3 Justification for Proposed Cache Architectures

In this section, we have first analyzed the influence of data block size on various parameters that justifies the need for our proposed architecture.

7.3.1 Cache Block Utilization

Previous research had reported that in the absence of high spatial locality, a multiple word cache blocks which are of 64 bytes in size on existing CMP tend to increase cache pollution and fill the cache with neighboring words that are unlikely to be used during block life time. To illustrate this issue, we divide the cache line into words of 8 bytes each and track which of the words are used before the complete block is evicted. The profiled results for few applications are shown in figure 7.3.

These results show that all of the executed applications accessed only 1-4 words within the complete 8 words (64bytes) more than 80 % of the total accesses.

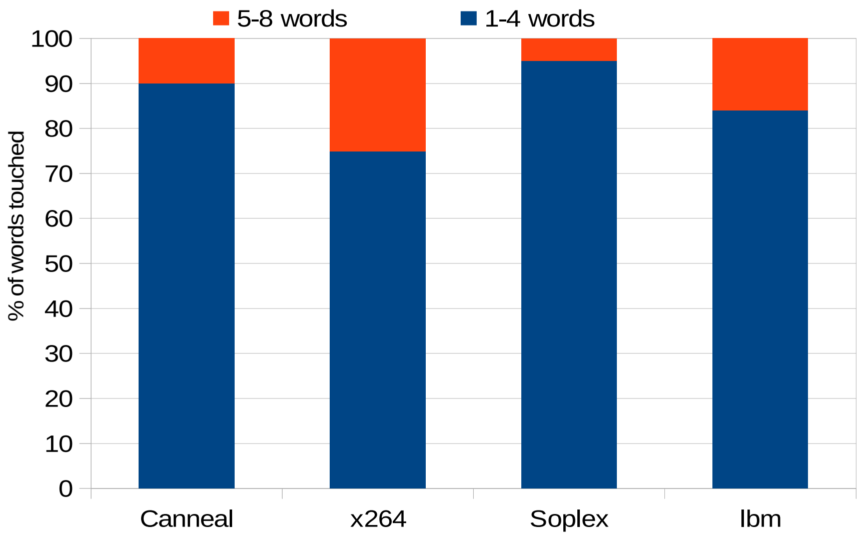


Figure 7.3: Percentage utilization of blocks

We have further analyzed the profiled results as shown in Figure 7.4. The result shows that all of the executed applications accessed 1-2 words on an average over 70% of times within 1-4 words (which is 80 % of the total accesses.)

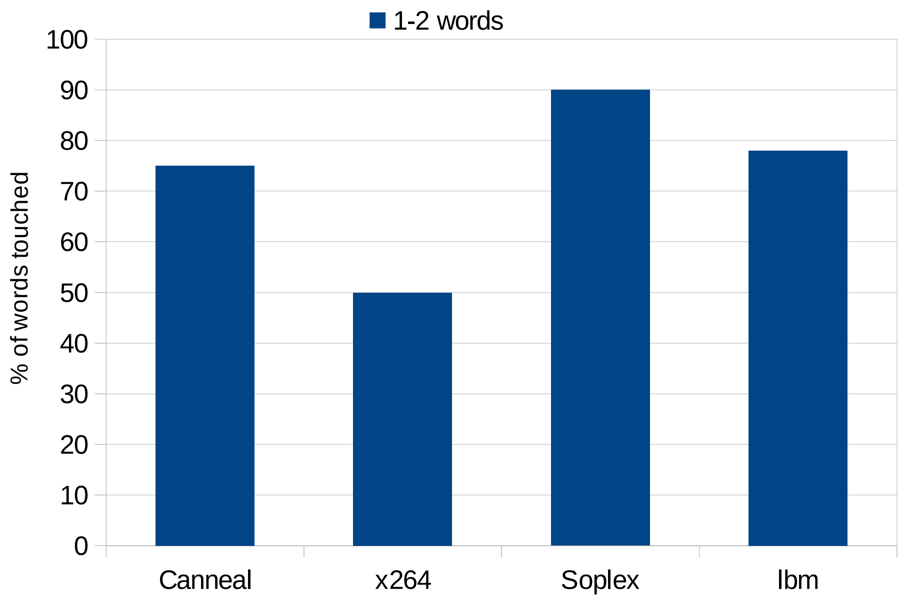


Figure 7.4: Percentage utilization of blocks

7.3.2 Effect of Block Size on Cache Miss Rate and Bandwidth

We have also observed that different applications have different cache block size requirements and it affect cache miss rate which is directly correlated with the performance and the size of the data transfer using interconnect network. The size of data transfer effects bandwidth and dynamic energy [144]. Our analysis shows that there is strong influence of block size on miss rate and bandwidth. We have executed an application on a system with 64K L1 cache and a 1M L2 cache with fixed ways in the cache with 64byte block size. In the next run we have reduced the block size from 64 to 32 bytes which increases the miss rate. However, when we have increased block size from 64 to 256 bytes, there is reduction in the miss rate but increase in the bandwidth. Therefore, there is a trade-off in miss rate and bandwidth and therefore choosing optimal block size presents the need to take both criteria's into account for an application.

7.3.3 Requirement for adaptive cache blocks

Previous research and our observation demands the need for novel cache architecture and hierarchy which supports variable cache block sizes that adapts the spatial locality of the data access patterns in an application. In summary: 1). A smaller fixed cache block improves cache utilization but it increases miss rate and interconnect traffic for applications with good spatial locality, affecting the overall performance. 2). A fixed Large cache block underutilizes the cache space and on chip interconnect with unused words for applications with low spatial locality, which significantly decreases the caching efficiency. 3). Spatial locality varies not only with applications but also within each application, resulting in underutilization of the significant fraction of the cache space.

In summary, a smaller fixed cache block can improve utilization and miss rate but is not suited for applications that exhibited good spatial locality. On the other hand, a large fixed size cache block goes under-utilized for applications exhibiting poor spatial locality. Since spatial locality varies both between applications as well as within an application, there exists a need for a cache which supports variable cache block sizes and adapts to the spatial locality of the data access patterns in an application.

We make the following contributions in this work:

1) Proposing a variable granularity cache, with variable size and number of cache blocks per set.

2) Designing the indexing, insertion, lookup and replacement polices for the proposed variable cache architecture.

3) Implementing the proposed policy in Verilog HDL and analyzing the results obtained.

The rest of the chapter is structured as follows: Section 7.4 provides detailed explanation of proposed architecture. Section 7.5 presents the details of the variable cache management scheme. Section 7.6 includes a discussion on the hardware overhead. Section 7.7 briefly outlines the spatial locality predictor used in the design. Results are presented in Section 7.8, with related work in Section 7.9 and concluding remarks in Section 7.10.

7.4 Proposed Variable Granularity cache architecture

Figure 7.5 shows the detailed architecture of our variable granularity cache. An important design consideration is the allocation of space for blocks (of different sizes) in the same cache according to the spatial locality shown by the application.

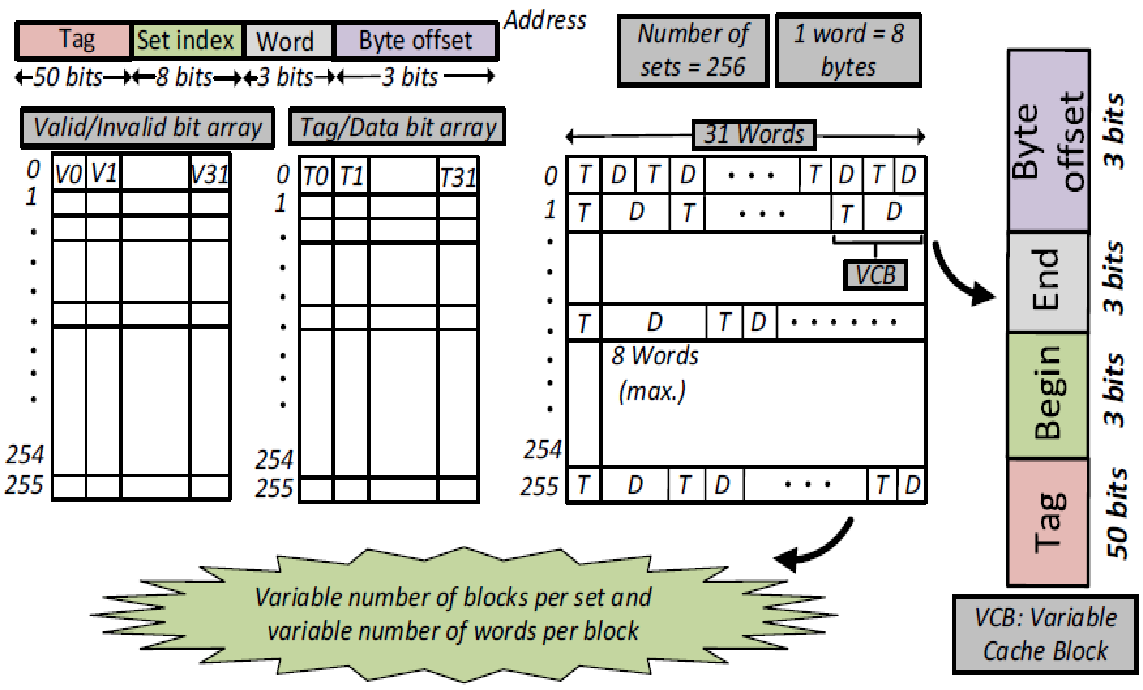


Figure 7.5: Variable Granularity Cache Architecture

Figure 7.5, presents a fixed size (64kB), variable granularity cache with 256 sets and 256 bytes per set. Cache sets may also be configured to contain: example eight 32-byte blocks or four 64-byte blocks, based on the application's requirement. As a result, our architecture is flexible both in terms of allowing for variable number of blocks as well as allowing blocks of different sizes in the same set. We refer to such a cache organization as a ‘locality-aware variable granularity cache organization.’

A traditional cache includes a separate data and tag array. But a cache architecture, such as ours would require the tag array to grow or shrink in size based on the number of blocks allocated to each set. One possible solution, is to completely eliminate the need for a separate tag array, by merging both the data arrays and tag arrays. This merging is beneficial because now, every tag co-located with the corresponding data. However this modification presents its own challenges. We describe these challenges below and outline solutions for the same.

1) **Distinguishing between data and tag words in the set**: One possible solution is the addition of a separate array to store bits to indicate which word in the cache array represents tags and which one represents data-blocks as shown in the Tag/data bit array in Figure 7.5.

2) **Tracking the validity of data stored in a cache line**: In a conventional cache, the valid/invalid bits are used by the block replacement and block insertion policies and are typically associated with the tag array. One possible solution is the addition of another array for storing information about the validity of data in every word of the set.

Finally as shown in Figure 7.5, the complete cache architecture has three separate arrays one for storing data-blocks and tags together, one for identifying tag/data words and one for storing the valid/invalid bit information. In the proposed cache architecture tags are extended with ‘Begin’ and ‘End’ addresses to support variable multi-word blocks. The next section gives the working of the overall cache architecture.

7.5 CACHE MANAGEMENT SCHEME

Some of the key aspects to be kept in mind while developing a cache architecture are the indexing, look-up, insertion and replacement policies. We describe them in detail below:

7.5.1 Cache Set-Indexing

In this cache architecture the main storage array holds a collection of sets with different sized data-blocks that do not overlap. Each cache block is divided into 4 different fields consisting of <Tag, Begin, End, Data-Block> as shown in Figure 7.5. The minimum size of the data block in the cache is one word and the maximum is TMAX words. The boundaries of any cache block are given by the ‘*Begin*’ and ‘*End*’ bits. We can encode ‘*Begin’* and ‘End’ in log2 (TMAX) bits. In the cache, the set indexing technique masks the lower log2 (TMAX) bits to ensure that all data-blocks in the same set index to the same set. The Tag and Set-

Index are identical for every word in the cache block. When TMAX = 8 words = 64 bytes, a fair comparison with a fixed block size (64 bytes) conventional cache architecture can be made.

7.5.2 Data Lookup

The steps involved in data look-up are described in Figure 7.6. In the first step, the lower log2 (TMAX) bits are masked from the address and the set index is derived from the remaining bits. In parallel, the Tag/Data bit array activates the words in the data array corresponding to the tags for comparison. In this cache architecture the minimum size of a block is two words, one for the tag, and one for the data, therefore adjacent words cannot be tags.

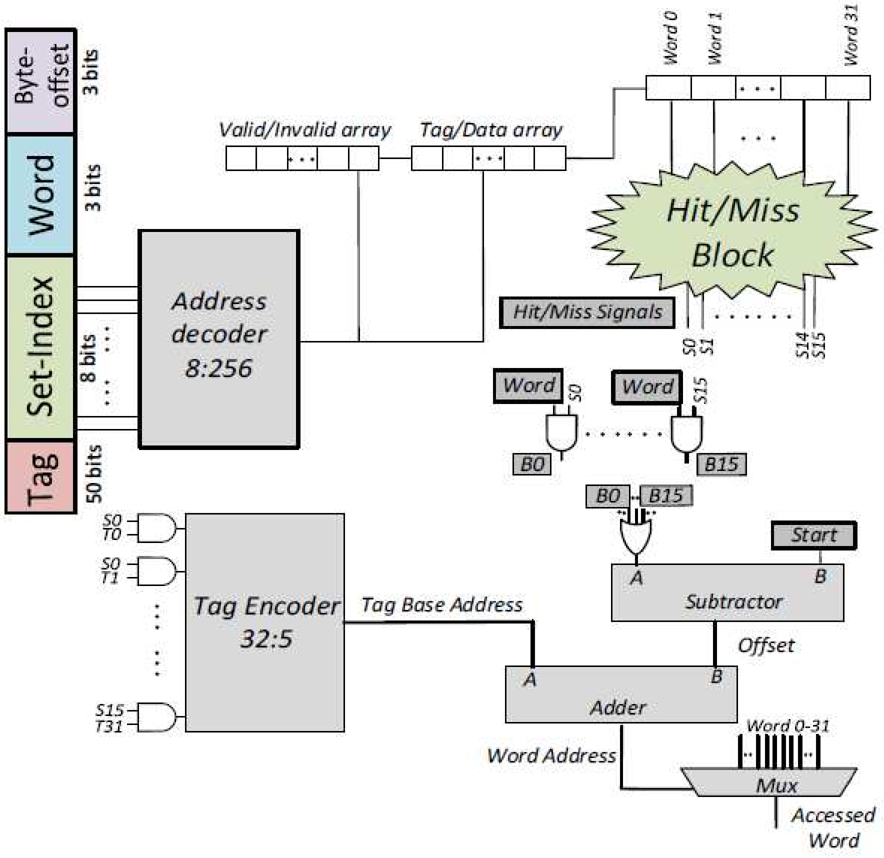


Figure 7.6: Data look-up logic

The hit-miss block shown in Figure 7.7 consists of two comparators, one to determine if there is a tag match and another one to ascertain if the requested word lies in the range specified by the ‘Begin’ and ‘End’ bits. Using the base address of the requested word from the tag encoder and the offset computed using a subtractor, we obtain the location of the requested word which is routed to the destination using a multiplexer.

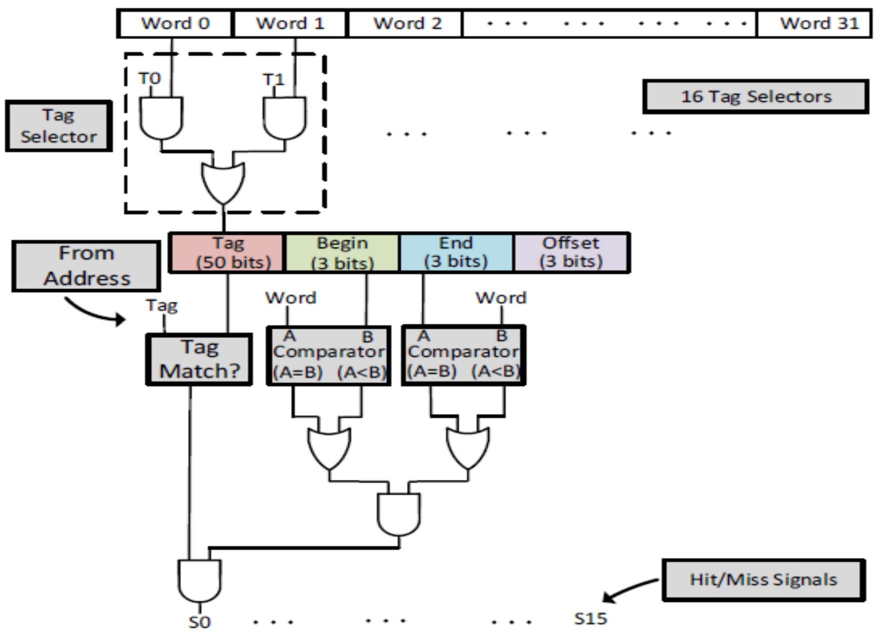


Figure 7.7: Hit/Miss Block

7.5.3 Block Insertion

In case of a miss for the desired word, the insertion policy should determine a position in the set to allocate the incoming block. In order to accomplish data insertion we examine the Valid/Invalid bit array. As described in Section 7.4 there is one bit per word in this array and a “1” in the bit field indicates that the corresponding word (tag/data) has been allocated space and contains valid information. So in order to reduce search space for an incoming data block, this architecture performs a substring search on the Valid/Invalid bit array of the cache set for contiguous sequence of “0s” (empty words). For example, to insert a block of four words consisting of a single word-sized tag and triple word-sized data, it performs a substring search for 0000 in Valid/Invalid bit array corresponding to the indexed set. In case a match is found, the tag and data block may be inserted and the corresponding bits in the tag/data bit array as well as in valid/invalid bit array can be set. However if the search results in a miss it triggers the replacement policy as described below in Section 7.5.4.

7.5.4 Block Replacement

The key challenge in this policy is to identify the block to replace. When the selected block in the cache is replaced, the corresponding bits in the tag/data bit array and valid/invalid bit array are reset. We employ Least Frequently Used (LFU) as our replacement policy as adopted in literature. It works as follows: Firstly, in the absence of vacant space in the cache set, the least frequently used (LFU) block is probed for replacement. If after such a replacement, there is still inadequate space available for the incoming block, our policy resorts to replacing multiple smaller blocks (based on their position in the least frequently used stack) till the incoming block may be accommodated.

7.6 Hardware Overhead

The extra bits required in this proposed cache are the tag/data bit per word and valid/invalid bit per word in both arrays. Both the tag/data bit array and valid/invalid bit array sizes are directly proportional to the cache size and require a minimal storage overhead of 3%.

7.7 Spatial Locality Predictor

A spatial locality predictor serves the purpose of determining the number of words to be fetched on a cache miss. We examine the execution traces from different applications and predict its spatial locality. In this work, we demonstrated the effectiveness of the technique for a custom trace. Later, we will extend this work, for any application phase. For this purpose, we intend to use a prediction table (similar to a branch history table) whose entries are indexed by the program counter (PC). Each entry of the table contains a bit array, whose field indicates whether a particular word has been touched before eviction. We use the PC to index into the table, based on the notion that specific PC’s capture the spatial locality of the application. The entries in the table need to be updated only on an eviction (often infrequent), hence, the additional latency that will be imposed by the predictor is minimal. Our predictor is optimistic and will over-fetch around the critical word requested by the processor. One may also choose to bypass the predictor (cold miss) when prediction accuracy is low (low confidence interval). Further, we also wish to conduct a sensitivity study to tackle certain other issues that come with online prediction, such as determining the optimum size of the prediction table and the prediction table entries, as a part of future work.

7.8 Results

The proposed architecture is simulated using Verilog Hardware Description Language (HDL) using ModelSim.

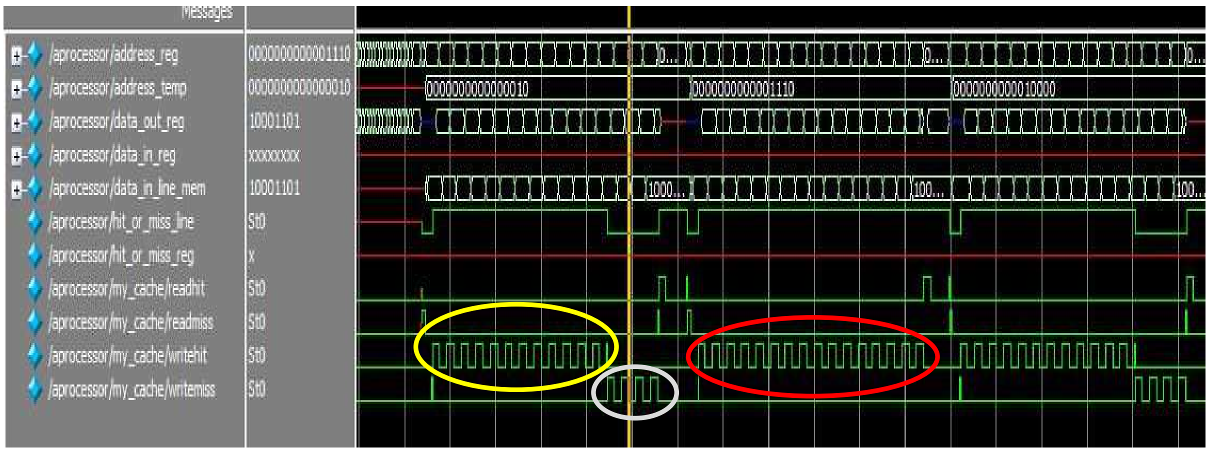


Figure 7.8: Read and Write Accesses to a set containing two blocks of size 3 and 4 words respectively

Figures 7.8 and 7.9 show how write accesses to different words in the variable cache are handled. We present accesses to two sets each containing variable sized blocks for the purpose of evaluation. In Figure 7.8, accesses to a single set of the variable granularity cache are shown along with various control signals. This set contains a block of size 3 words (12 bytes) and another of size 4 words (16 bytes). We assume that a single byte transaction occurs per cycle. The yellow oval indicates 12 write hits to the 3-word block. However a request for a subsequent word results in a write miss as shown by the gray oval. The spatial locality predictor predicts a complete 4-word block (TMAX) be used to refill the set. The 3-word block by virtue of being least frequently used (due to previous accesses) is evicted. Since there is still inadequate space available, the 4-word block is evicted as well. Now, as indicated by the red oval, 16 write hits corresponding to the incoming block may be noticed. As a result of the sequence of operations, a 3-word void is left in the set.

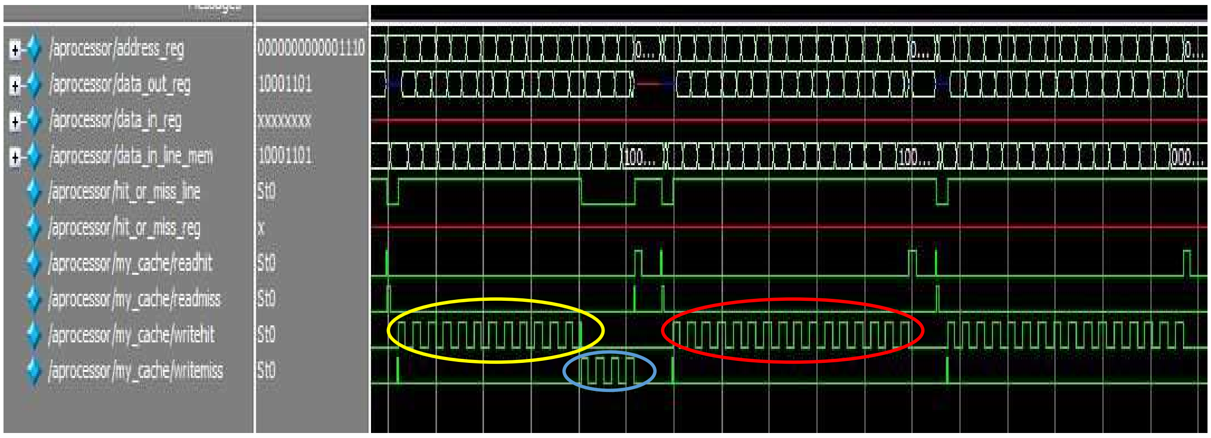

Messages
/aprocessor/address_reg
/aprocessor/data_out_reg
/aprocessor/data_in_reg
/aprocessor/data_in_line_mem
/aprocessor/hit_or_miss_line
/aprocessor/hit_or_miss_reg
/aprocessor/my_cache/readhit
/aprocessor/my_cache/readmiss
/aprocessor/my_cache/writehit
/aprocessor/my_cache/writemiss
0000000000001110
10001101
xxxxxxxx
10001101
St0
x
St0
St0
St0
St0

Figure 7.9: Read and Write Accesses to another set containing two blocks of size 3 and 2 words respectively.

The same sequence of operations as described earlier takes place, with the difference that the 2-word block is evicted instead of the 4-word block. In this case, assuming the same predictor is used, a single word void is left in the set. In Figure 7.9, accesses to another set containing a 3-word and 2-word block are shown, whereas figure 7.10 shows complete cache simulation.

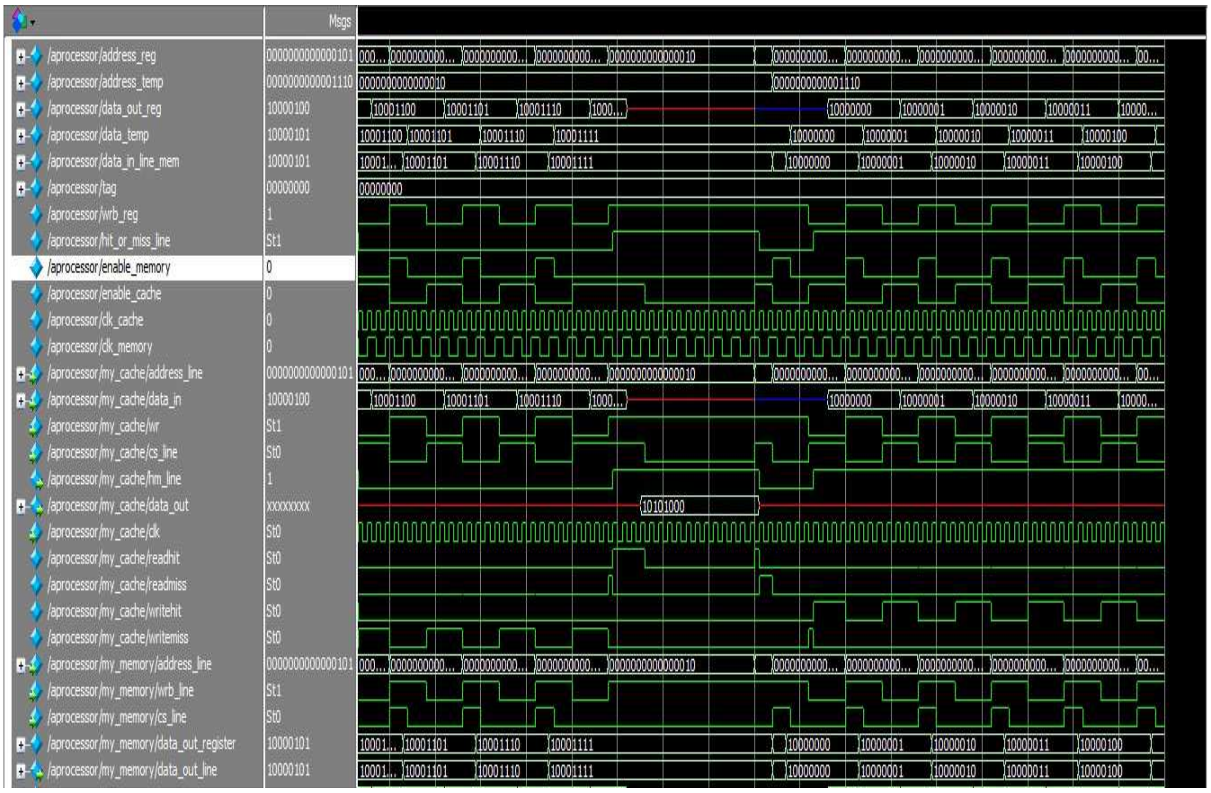

Msgs
/aprocessor/address_reg
/aprocessor/address_temp
/aprocessor/data_out_reg
/aprocessor/data_temp
/aprocessor/data_in_line_mem
/aprocessor/tag
/aprocessor/wrb_reg
/aprocessor/hit_or_miss_line
/aprocessor/enable_memory
/aprocessor/enable_cache
/aprocessor/clk_cache
/aprocessor/clk_memory
/aprocessor/my_cache/address_line
/aprocessor/my_cache/data_in
/aprocessor/my_cache/wr
/aprocessor/my_cache/cs_line
/aprocessor/my_cache/hm_line
/aprocessor/my_cache/data_out
/aprocessor/my_cache/clk
/aprocessor/my_cache/readhit
/aprocessor/my_cache/readmiss
/aprocessor/my_cache/writehit
/aprocessor/my_cache/writemiss
/aprocessor/my_memory/address_line
/aprocessor/my_memory/wrb_line
/aprocessor/my_memory/cs_line
/aprocessor/my_memory/data_out_register
/aprocessor/my_memory/data_out_line

Figure 7.10: Cache Simulation.

7.9 Related Work

There has been a large body of research working on improving the utilization of the cache and reduce energy consumption [143][145][148]. Qureshi et al. [151] proposed Line Distillation to discard only untouched words from a block during eviction. Their design consists of a Line Organized Cache (LOC) in which the cache blocks are of regular granularity (64 bytes) and a Word Organized Cache (WOC) which contains word-sized blocks. Therefore, this organization supports the storage of data at two granularities in the cache. In contrast, we propose to maintain different word-sized cache blocks in order to fully exploit the spatial locality shown by the application. Veidenbaum et al. [140] also proposed a word-organized cache, but it incurs significant tag overhead. Sector caches have also been proposed in the past [141][142][147]. They organize tags at the granularity of a sector and data at sub-sector granularity. In particular, Pujara et al. [144] proposed a word-sized sector cache that uses prefetching to determined words that may be utilized by the application. But a common problem faced by all prefetching techniques is the issue of cache pollution with unused words. Orthogonal to the above mentioned techniques, work has also been done towards designing an adaptive granularity DRAM based architecture [146]. Similarly, many techniques have been proposed at the software level (compiler) to re-order code to better exploit spatial locality [149] [150].

7.10 Summary

In this chapter, a locality-aware variable granularity cache architecture is presented, that can hold different number of cache blocks with variable number of words. This adaptive block sizing minimizes the size of data messages and reduces on chip network traffic. By utilizing a spatial locality predictor, we are able to reduce cache pollution for applications that exhibit low spatial locality and improve the performance of other applications. Our variable granularity cache is flexible and can be adapted to suit any level of the multilevel cache hierarchy (L1, L2 or L3). We have used this novel cache to model L1 in the cache hierarchy. Simulations in Verilog HDL demonstrate the feasibility of the proposed design. In future, we will perform full system simulations using cache simulation tools [99]. In addition, the number of words per block utilized by the application will also be evaluated by profiling the cache evictions for a variety of benchmarks [108]

Chapter 8

Conclusions and Future Work

This chapter presents the main conclusions of this thesis and outlines areas for future work.

Chapter 8

Conclusions and Future work

In this chapter, we conclude the thesis by summarizing the contributions and providing some future directions for extending the work.

8.1 Conclusions

In order to take advantage of billions of transistors on single chip with manageable design complexity while staying within the power budgets and meeting the demand of ever increasing throughput requirement, CMP with many cores and shared LLC is a viable design choice. With the adoption of this domain, we have higher demand on on-chip cache capacity and interconnect bandwidth (on/off-chip). Many multi-threaded applications on CMP require support for fine-grain and dynamically changing sharing access patterns. Multiprogrammed and single-threaded applications require localized data access. All these applications are penalized by indirection in directory-based cache coherence. Furthermore, their working sets well exceed the private cache sizes and stress-test the shared LLC mostly by exceeding the on-chip capacity. On-chip caches must therefore adapt to these varying needs to reduce L1 miss penalties and both on chip and off-chip bandwidth requirements.

In this thesis, we have tried to incorporate different cache management schemes to design a CMP cache that solves the indirection problem as well as meets the requirements of fine-grain sharing support, localized and faster coherence and data availability, larger effective cache capacity, and application-adaptive replacement-migration policy.

As stated above, most of today’s multi-core processors feature Last level shared L2 caches. A major problem faced by such multi-core architectures is cache contention, where multiple cores compete for usage of the single shared L2 cache. Previous research shows that uncontrolled sharing leads to scenarios where one core evicts useful L2 cache content belonging to another core. To address this problem:

We examined in Chapter 4 a cache miss classification – CII: Compulsory, Inter-processor and Intra-processor misses – for CMPs with shared caches and its comparison to 3C miss classification for traditional uniprocessor, to provide a better understanding of the interactions between memory references of different processors at the level of shared cache in a CMP. We then propose a novel approach, called block pinning, for eliminating inter-processor misses and reducing intra-processor misses in a shared cache. Further, we showed that an adaptive block pinning scheme improves over the benefits obtained by the block pining and set pinning scheme by significantly reducing the number of off–chip accesses. This work also proposes two different schemes of relinquishing the ownership of a block to avoid domination of ownership of few active cores in multicore system which results in performance degradation. Extensive analysis of these approaches with SPEC and Parsec benchmarks are performed using a full system simulator.

In Chapter 5 we presented the growing needs of modern memory-hungry work-loads, therefore there is a growing need to keep large size on-chip caches. Unfortunately, expanding the cache size alone is not sufficient to increase modern systems efficiency, since the traditional UCA design exhibits serious limitations, larger capacity comes at the cost of increased access latency, as wire delays grow along with the physical size of the memory structure. For that reason, large on-chip caches with a single, large and uniform latency are undesirable**.** In other words, increasing cache sizes only makes the existing gap between processor and memory access speeds grow even wider. The solution lies in a distributed cache design that manages to provide varying access times and increased bandwidth. In order to achieve this goal, a complete shift in the cache architecture design paradigm was required. The previously single, monolithic chunk of cache (UCA) is transformed to a finer-grained structure. More specifically, the last-level cache is composed of physically independent *banks*, which are evenly distributed across the die area. This design provides varying access latencies between the cores and the cache banks, depending on the physical distance between the requesting core and the cache bank where the requested data resides. Thus, we are led to a Non-Uniform Cache Access (NUCA) organization. NUCA provides faster access to cache blocks in the banks that reside closer to the processor. The major limitation with this architecture is that a block can only be placed in a single location during its lifetime. This, of course, imposes serious limitations with this architecture: a frequently accessed block may be placed in a bank located far from the cache controller, thus suffering the overhead of a high access time every time it is accessed. We proposed an efficient, and low-overhead mechanism to track the re-usability of each cache line in the shared NUCA. Our scheme allows dynamic replication of those cache lines that shows high usage at the shared LLC. When a replicated cache line is evicted or invalidated, the proposed scheme dynamically adjusts its future replication decision. This scheme also reduces access latency and energy consumption by selectively replicating the cache line that show high re-usability in the local bank-cluster of the requesting core. It also maintains coherence complexity similar to that of a conventional non-hierarchical coherence protocol as replications are allowed only in the local bank cluster of the requesting core.

Chapter 6 of this thesis dealt with the challenges raised by Dynamic NUCA design. The limitations of the static NUCA organization resulted in NUCA's next generation designs, the dynamic NUCA, which address the problems that rise from static placement. Furthermore, future multi-core systems will execute massive memory intensive applications with significant data sharing. Data movement and their management further impacts memory access latency and consumes power. We observed that previous D-NUCA designs have used a costly data access scheme to locate data in the NUCA cache in order to achieve remarkable performance improvement. To address these situations, we further investigated this limitation along with the benefits of dynamic NUCA organization and also discussed the drawbacks of both S-NUCA and D-NUCA organization. Finally, we proposed an adaptive migration-replication policy for non-uniform shared last level cache and proposed an efficient data access policy using a set of location pointers with each banks, which addresses the basic problems with these two potential future cache architectures SNUCA and D-NUCA. Our scheme relies on low-overhead and highly accurate in-hardware pointers to control network traffic and improves cache miss latency. Using simulations on 8-core multi-core system, we show that our proposed data search mechanism in D-NUCA design reduces dynamic energy consumed per memory request and outperforms multi cast access policy by an average performance speedup.

In Chapter 7 we first presented the need for hybrid novel cache design based on the observation of variable spatial locality exits among different application. Then, we presented a novel cache architecture with adaptive block sizing to minimize the size of data movement and reduces on chip network traffic.

To summarize, we optimized for both private and shared data in all types of applications. We optimized for shared data in multi-threaded applications by providing fair adaptive block ownership policy and its dynamic relinquishment (at block level).

We tracked frequency of usage of data in all types of applications including multiprogrammed, multi-threaded applications on the fly and triggers selective replication of most frequently used data at the local bank cluster and localized coherence in NUCA (Chapter 5).

We also optimized cache for all types of applications by preventing data ping-pong and uncontrolled data movements within NUCA using adaptive migration-replication (AMR) policy (Chapter 6).

Our optimizations are applied at the L1 level using fine-grain variable size block movement from LLC/L2 level (larger effective L1 capacity, as more words are moved close to private L1 cache) (Chapter 7).

8.2 Future Directions

The experimental work presented in this thesis opens up following directions in the cache hierarchy and coherence protocol design:

8.2.1 Global Replacement Policy.

Current last level non-uniform cache architectures (NUCA) for multicore processors employ LRU (Least Recently used), PLRU (Pseudo-LRU) or its variants as their replacement strategy. These policies work well for a traditional uniform cache architecture but none of them address the issue of global cache line replacement as required in a heavily banked NUCA cache. In a NUCA cache, highly reused cache lines placed in the local banks (near the requesting cores) which face frequent eviction as compared to cache lines are placed far away. This can lead to increased miss rates for different applications. A conventional replacement policy employed at the local bank evicts the LRU cache line, without considering the possibility of its future use. This policy also does not consider idle cache lines (showing lesser reuse) at distant banks as candidates for replacement. Since multiple banks in a NUCA cache work independently, there exists no means to identify the LRU cache line at a global level, considering all banks. Therefore, there is a need for a global cache replacement scheme that characterizes cache lines based on their reuse probability, and prioritizes the retention of those blocks showing high reuse probability.

8.2.2 Dynamic granularity block movement with Coherence Granularity for caches in CMP

Current research proposals and existing work maintains cache coherence at cache line granularity or at page level granularity. With this fixed line/page size, it is easy to design and maintain cache coherence in CMP. However, the main limitation of these proposals are that, they do not allow to change the granularity of the line/page dynamically depending on the workload pressure. Different workload have variable line size requirements and hence variable granularity cache line has the potential to improve the overall performance in CMP.

8.2.3 Mapping strategy:

Conventional, static cache line and page mapping to the multi-banked last level cache banks has the benefit of easy implementation. However, they do suffer from the long access latency due to initial poor placement. In future, an efficient mapping policy is required along with variable granularity block and cache coherence support.

8.2.4 Tiled architecture:

Analysis of our proposed schemes on a tiled architecture will be another interesting area of our future work.

LIST OF REFERENCES

[1] G. E. Moore. Cramming More Components onto Integrated Circuits. Electronics, pp. 114–117, April 1965.

[2] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future cmps. In Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 199-210, 2001

[3] K. Krewell. Intel's PC roadmap sees double. Microprocessor Report, vo1.8, issue 5, pp. 41-43, May 2004.

[4] R. Low. Microprocessor trends: multicore, memory, and power developments. Embedded Computing Design, Magzine, September, 2005.

[5] M. Monchiero, R. Canal, and A. Gonzlez, Power/Performance/Thermal Design-Space Exploration for Multicore Architectures. In IEEE Transactions on Parallel and Distributed Systems, vol. 19, issue 5, pp. 666–681, 2008

[6] Frank Schirrmeiste. Multi-core Processors: Fundamentals, Trends, and Challenges. Embedded Systems Conference, Imperas, Inc., 2200, pp. 6-15, California, April 4, 2007.

[7] Christian Martin. Multicore Processors: Challenges, Opportunities, Emerging Trends. In proceeding of the Embedded world Exhibition and Conference, pp. 1-6, February 2014.

[8] Ardsher Ahmed, Pat Conway, Bill Hughes, and Fred Weber. AMD Opteron™ shared-memory MP systems. In Proceeding of the 14th Hot Chips Symposium, pp. 1-30, August 2002.

[9] Chetana N. Keltcher Kevin J. McGrath Ardsher Ahmed Pat Conway. The AMD opteron processor for Multiprocessor Server. IEEE Micro, vol. 23, issue 2, pp. 66-76, March/April, 2003.

[10] Ron Kalla Balaram Sinharoy Joel M. Tendler, IBM Power5 Chip: A multithreaded Dual core processor, IEEE Micro, vol. 24, issue 2, pp. 40-47, March/April, 2004.

[11] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. IEEE Micro, vol. 25, issue 2, pp. 10-20, March-April 2005.

[12] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, January 2011.

[13] K. Krewell. Sun's Niagara pours on the cores. Microprocessor Report, vol. 18, issue 9, pp. 11-13, September 2004.

[14] Borkar, S., Chien, A. A. The Future of Microprocessors. Communications of the ACM, Vol. 54, No. 5, pp. 67-77, May 2011.

[15] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In 27th International Symposium on Computer Architecture (ISCA), pp. 248–259, June 2000.

[16] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. In IEEE transaction, vol. 89, issue 4, pp. 490–504, April 2001.

[17] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In Proceedings of the 7th International Conference on

Architectural Support of Programming Languages and Operating Systems, Cambridge, MA, pp. 2-11, October 1996.

[18] P. Kongetira, K. Aingaran, and K. Olukotun, Niagara: A 32-Way Multithreaded SPARC Processor, IEEE Micro, vol.25, issue 2, pp. 21–29, 2005.

[19] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 211-222, 2002.

[20] A. J. Smith. Cache memories. Computing Surveys, vol. 14, issue 3, pp. 473–530, September 1982.

[21] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. ACM Transaction on Design Automation Electronic System, vol. 5, issue 2, pp. 115–192, Apr. 2000.

[22] L. Benini. Energy-Aware Design of Embedded Memories : A Survey of Technologies, Architectures, and Optimization Techniques, vol. 2, issue 1, pp. 5–32, 2003.

[23] P. R. Panda, F. Catthoor, K. U. Leuven, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and Memory Optimization Techniques for Embedded Systems. ACM Transaction on Design Automation of Electronics Systems, vol. 6, issue 2, pp. 149–206, 2001.

[24] R. T. Witek, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, G. W. Hoeppner, T. H. Lee, P. C. M. Lin, L. Madden, M. H. Pearce, K. J. Snyder, and S. C. Thierauf, “0. 5-W CMOS RISC,” vol. 9, issue 1, pp. 1703–1714, 1997.

[25] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc., 4th edition, 2007.

[26] S. J. E. Wilton and N. P. Jouppi. CACTI : An Enhanced Cache Access and Cycle Time Model. IEEE Journal of solid state circuits, pp. 1–26, 1996.

[27] T. Chen and J. Baer. Effective hardware-based data prefetching for high-performance processors. IEEE Transaction on Computing, vol. 44, issue 5, pp. 609–623, May 1995.

[28] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. In IEEE Transactions on Computers, vol. 38, issue 12, pp. 1612–1630, December 1989.

[29] A. J. Smith. Cache Memories,” ACM Computer. Survey, vol. 14, issue 3, pp. 473–530, September 1982.

[30] J. Jeong and M. Duhois. Optimal Replacements in Caches with Two Miss Costs. In Proceedings of the 11th Annual ACM symposium on Parallel Algorithms and Architectures, pp. 155–164, June 1999.

[31] T. S. B. Sudarshan, R. A. Mir, and S. Vijayalakshmi. Highly Efficient LRU Implementations for High Associativity Cache Memory. In Proceedings of the 12th IEEE International Conference on Advanced Computing and Communications, pp. 87-95, December 2014.

[32] W. Wong and J. Baer. Modified lru policies for improving second-level cache behavior. In Proceedings of the 6th International Symposium on High-Performance Computer Architecture, pp. 49-60, January 2000.

[33] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In Proceedings of the 42nd Annual Southeast Reg. Conference, ACM-SE 42, pp. 267-272, 2004.

[34] Y. Deville and J. Gobert. A Class of Replacement Policies for medium and high associativity Structures. ACM SIGMETRICS Computer Architecture News, vol. 20, issue 1, pp. 55-64, March 1992.

[35] R. A. Sukumar and S. G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Application to Miss Characterization. In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer System, pp. 24-35, May 1993.

[36] M. Kampe, P. Stenstrom, and M. Dubois. Self-correcting LRU replacement policies. In Proceedings of the first Conference on Computing Frontiers, pp. 181-191, April 2004.

[37] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU : Simple and Effective Adaptive Page Replacement. In Proceedings of the 1999 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, pp. 122–133, May 1999.

[38] A. V. Veidenbaum, W. Tang, R. Gupta, R. Nicolau and X. Ji. Adaptive Cache line Size to Application Behavior. In Proceedings of the 13th International Conference on Supercomputing, pp. 145-154, June 1999.

[39] N. Maki, K. Hoson and A. Ishida. A Data-Replace-Controlled Cache Memory System and its Performance Evaluations. In Proceedings of the IEEE Region 10 Conference, pp. 471–474, September 1999.

[40] Z. Wang, and D. O. F. Philosophy. Cooperative hardware/software caching for next generation memory systems. Ph.D Thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, February, 2004.

[41] E. J. O'Neil, P. E. O'Neil, and G. Weikum. An optimality proof of the LRU-K page replacement algorithm. Journal of ACM, vol. 46, issue 1, pp. 92–112, Jan, 1999.

[42] A. Lai. C. Fide and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. ACM SIGARCH Computer Architecture News, vol. 29, issue2, pp. 144–154, 2001.

[43] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 134-142, May 1990.

[44] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho and C. S. Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. In IEEE Transaction on Computers, vol. 50, issue 12, pp. 1352–1361, December. 2001.

[45] Y. Smaragdakis. General adaptive replacement policies. In Proceedings of the 4th International Symposium on Memory Management - ISMM '04, pp. 108-119, October, 2004.

[46] J. Alghazo, A. Akaaboune, and N. Botros. Sf-lru Cache Replacement Algorithm. In Proceedings of the Records 2004 International Workshop on Memory Technology, Design and Testing, pp. 19-24, August 2004.

[47] J. Aguilar and E. L. Leiss. An Adaptive Coherence-Replacement Protocol for Web Proxy Cache Systems. Communication and Systems, vol. 8, pp. 1–14, 2004.

[48] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In Proceedings of the 17th Annual International Symposium on Computer Architecture, vol. 18, issue 3, pp. 364-373, May, 1990.

[49] M. Flynn and P. Hung. Microprocessor design issues: Thoughts on the road ahead. IEEE Micro, vol. 25, issue 3, pp.16–31, 2005.

[50] A lebeck, X. Fan, H. Zeng and C. Ellis. Power-Aware Page Allocation. ACM SIGOPS Operating Systems Review, vol. 34, issue 5, pp. 105-116, 2000.

[51] A. Acquaviva and B. Ricc. Energy Characterization of Embedded Real-Time Operating Systems. In Proceedings of the Workshop on Compliers and Operating Systems for Low Power, pp. 53-73, December 2003.

[52] A. Gutierrez, R. G. Dreslinski, T. Mudge. Evaluating Private vs. Shared Last-Level Caches for Energy Efficiency in Asymmetric Multi-Cores. In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), pp. 191-198, July, 2014.

[53] Li Zhao, Ravi Iyer, Mike Upton, Don Newell. Towards Hybrid Last Level Caches for Chip-Multiprocessors. Intel Corporation. 2006. Available from: http://wwwpassat.crhc.uiuc.edu/dasCMP/papers/dasCMP07/paper07.pdf.

[54] Moinuddin K. Qureshi Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance Runtime Mechanism to Partition Shared Caches. In Proceeding of the 39th Annual IEEE/ACM international Symposium on Micro-architecture, Orlando, Florida, USA, pp. 423-432, Dec.2006.

[55] Evan Speight, Hazim Shafi, Lixin Zhang and Ram Rajamony. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In Proceeding of the 32nd Annual International Symposium on Computer Architecture (ISCA'05), Madison, Wisconsin USA, pp. 346-356 June 2005.

[56] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, Christos Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. ACM SIGARCH Computer Architecture News New York USA, Volume 35, Issue 2, pp. 358-368, May 2007.

[57] Jichuan Chang and Gurindar S. Sohi, Cooperative Cache Partitioning for Chip Multiprocessors. In Proceedings of the 21st annual international conference on Supercomputing, Seattle, Washington, pp. 242-252, Dec 2007.

[58] K. T. Sundararajan, V. Porpodas, T.M. Jones, N.P. Topham, B. Franke. Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs. In Proceeding of IEEE 18th International Symposium High Performance Computer Architecture (HPCA), pp. 1-12, Feb. 2012.

[59] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell,Yan Solihin, Lisa Hsu, Steve Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In Proceedings of the ACM SIGMETRICS 2007 the International Conference on Measurement and modeling of Computer Systems, San Diego, pp. 23-24, June 2007.

[60] Bradford M. Beckmann Michael R. Marty and David A. Wood. ASR: Adaptive Selective Replication for CMP Caches**.** In Proceedings of the 39th Annual IEEE/ACM international Symposium on Micro-architecture (MICRO-39), Orlando, FL, pp. 443-454, Dec 2006.

[61] Seongbeom Kim, Dhruba Chandra and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'04), France, pp. 111- 122, Oct. 2004.

[62] Lei Jin Hyunjin Lee Sangyeun Cho. A Flexible Data to L2 Cache Mapping Approach for Future Multi-core Processors. In Proceedings of the 2006 workshop on Memory system performance and correctness, San Jose, California, pp. 92-101, Nov 2006.

[63] Jaehyuk Huh Changkyu Kim, Hazim Shafi Lixin Zhang Doug Burger Stephen W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In Proceeding of the 19th International Conference on Supercomputing, ICS 2005, Cambridge, assachusetts, USA, pp. 1028-1040, June 2005.

[64] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, Stephen W. Keckler "Author Retrospective for A NUCA Substrate for Flexible CMP Cache Sharing", ICS 25th Anniversary Volume, pp. 74-76, June, 2014.

[65] Pierfrancesco Foglia, Daniele Mangano, Cosimo Antonio Prete, NUCA Model for Embedded Systems Cache Design. IEEE 2005 Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA), New York Metropolitan Area, USA, pp. 41-46, September 2005.

[66] G.E. Suh, L. Rudolph, S. Devadas. Dynamic Partitioning of Shared Cache Memory. The Journal of Supercomputing, Volume 28, Number 1, pp. 7-26, April 2004.

[67] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez and Mateo Valero. Explaining Dynamic Cache Partitioning Speed Up. IEEE Computer Architecture Letter, vol. 6, issue 1, pp. 1-4, Jan 2007.

[68] Zvika Guz, Idit Keidar, Avinoam Kolodny, Uri C. Weiser. Nahalal: Cache Organization for Chip Multiprocessor. IEEE Computer Architecture Letters, vol. 6, issue 1, pp. 21-24, Jan 2007.

[69] Alexandra Fedorova, Margo Seltzer and Michael D. Smith. Cache-Fair Thread Scheduling for Multicore Processors. Technical Report TR-17-06, Division of Engineering and Applied Sciences, Harvard University Cambridge, October 2006.

[70] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing Shared L2 Caches on Multicore Systems in Software. Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), held in conjunction with the International Symposium on Computer Architecture (ISCA), Toronto, Canada, Jan 2007.

[71] Hari Kannan, Fei Guo, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, Christos Kozyrakis. From Chaos to QoS: Case Studies in CMP Resource Management. ACM SIGARCH Computer Architecture News, New York, USA, vol. 35, issue 1, pp. 21-30, June 2007.

[72] H. Kasture and D. Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In Proceedings of the 19th international conference on

Architectural support for programming languages and operating systems (ASPLOS), March, 2014.

[73] Afrin Naz, Mehran Rezaei, Krishna Kavi and Philip Sweany. Improving Data Cache Performance with Integrated Use of Split Caches, Victim cache and Stream Buffers. Media Workshop 04, ACM SIGARCH Computer Architecture News, New York, USA, pp. 41-48, Nov 2005.

[74] Jie Tao, Marcel Kunze, and Wolfgang karl. Evaluating the cache Architecture of Multicore processors. In Proceeding of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, France, pp. 12-19, February 2008.

[75] Roy A., Vadlamani S., Sudarshan T.S.B. Variable Forwarding Cache Coherency for Chip Multiprocessors. In Proceeding of 14th Annual International Conference on High Performance Computing (HiPC 07), pp. 40-45, October 2007.

[76] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. IEEE Computer, vol. 29, issue 12, pp. 66–76, December 1996.

[77] L. Lamport. How to Make a Multiprocess Computer that Correctly Executes Multiprocess Programs. In IEEE Transactions on Computers, pp. 690–691, 1979.

[78] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 15–26, May 1990.

[79] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In 10th International Symposium on Computer Architecture (ISCA), pp. 124–131, June 1983.

[80] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. In IEEE Transactions on Computers, vol. 23, issue 6, pp.12–24, June 1990.

[81] Per Stenstrom, Mats Brorsson, Fredrik Dahlgren, Hakan Grahn, and Michel Dubois. Boosting the performance of shared memory multiprocessors. IEEE Transactions on Computers, vol. 30, issue 7, pp.63–70, July 1997.

[82] Hakan Nilsson and Per Stenstrom. An adaptive update-based cache coherence protocol for reduction of miss rate and traffic. In Proceedings of the 6th International Conference on Parallel Architectures and Languages Europe (PARLE), pp. 363–374, June 1994.

[83] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 414–423, June 1986.

[84] Alberto Ros, Manuel E. Acacio, and José M. García. An efficient cache design for scalable glueless shared-memory multiprocessors. In Proceedings of the ACM International Conference on Computing Frontiers, pp. 321–330, May 2006.

[85] Alberto Ros, Ricardo Fernández-Pascual, Manuel E. Acacio, and José M. García. Two proposals for the inclusion of directory information in the last-level private caches of glueless shared-memory multiprocessors. Journal of Parallel Distributed Computing (JPDC), vol. 68, issue 11, pp. 1413–1424, November, 2008.

[86] A. Charlesworth, Starfire: Extending the SMP Envelope, IEEE Micro vol. 18, issue 1, pp. 39–49, February 1998.

[87] A. Charlesworth. The Sun Fireplane System Interconnect. In Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, p-7-15, November 2001,

[88] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In Proceedings of the 30th Annual International Symposium on Computer Architecture, pp. 182–193, June 2003,

[89] Anant Agarwal, Richard Simoni, John L. Hennessy, and Mark A. Horowitz. An evaluation of directory schemes for cache coherence. In Proceedings of the 15th International Symposium on Computer Architecture (ISCA), pages 280–289, May 1988.

[90] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multi-cache systems. In IEEE Transactions on Computers, vol. 27, issue 12, pp. 1112–1118, December 1978.

[91] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark A. Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. In IEEE Computer, vol. 25, issue 3, pp. 63–79, March 1992.

[92] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark A. Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The Stanford FLASH multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture (ISCA), pp. 302–313, April 1994.

[93] James Laudon and Daniel Lenoski. The SGI Origin: A cc-NUMA Highly Scalable Server. In Proceedings of the 24th International Symposium on Computer Architecture (ISCA), pp. 241–251, June 1997.

[94] Kourosh Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and Design of AlphaServer GS320. In Proceedings of the 9th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS), pp. 13–24, November 2000.

[95] Luiz A. Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In Proceedings of the 27th International Symposium on Computer Architecture (ISCA), pp. 12–14, June 2000.

[96] Manish Shah, Jama Barreh, Jeff Brooks, Robert Golla, Gregory Grohoski, Nils Gura, Rick Hetherington, Paul Jordan, Mark Luttrell, Christopher Olson, Bikram Saha, Denis Sheahan, Lawrence Spracklen, and Aaron Wynn. 236 UltraSPARC T2: A highly-threaded, power-efficient, SPARC SoC. In IEEE Asian Solid-State Circuits Conference, pp. 22–25, November 2007.

[97] Manuel E. Acacio, José González, José M. García, and José Duato. A new scalable directory architecture for large-scale multiprocessors. In Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA), pp. 97–106, January 2001.

[98] Yeimkuan Chang and Lasimi N. Bliuyan. An efficient hybrid cache coherence protocol for shared memory Multiprocessors. IEEE Transactions on Computers, pp. 352–360, March 1999.

[99] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J.Högberg, F. Larsson, A.Moestedt, and B.Werner. Simics: A Full System Simulator Platform, vol. 35, issue 2, pp. 50–58, 2002.

[100] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. In Computer Architecture News, pp. 92-99, 2005.

[101] Niket Agarwal, Li-Shiuan Peh, and Niraj Jha. Garnet: A detailed interconnection network model inside a full-system simulation framework. Technical Report CE-P08-001, Princeton University, pp. 33-42, 2008.

[102] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to understand large caches. Technical report, University of Utah and Hewlett Packard Laboratories, 2007.

[103] A. Bardine, P. Foglia, G. Gabrielli, and C. A. Prete. Analysis of static and dynamic energy consumption in nuca caches: Initial results. In Proceedings of the Workshop on Memory Performance: Dealing with Applications, Systems and Architecture, pp. 105-112, 2007.

[104] H. S. Wang, X. Zhu, L. S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In Proceedings of the 35th International Symposium on Microarchitecture, pp. 294-305, 2002.

[105] Micron. System power calculator. In http : //www.micron.com/, 2009.

[106] Benchmarks. Spec cpu2006. In http : //www.spec.org/cpu2006, 2006.

[107] C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, Methodological considerations and Characterization of the SPLASH-2 Parallel Application Suite. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24–36, June 1995.

[108] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 72-81, 2008.

[109] S. Srikantaiah, M. Kandemir, M. J. Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 135-144, March 2008.

[110] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. "Exploring the cache design space for large scale cmps." SIGARCH Computer Architecture News, vol. 33, issue 4, pp. 24–33, 2005.

[111] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The detection and elimination of useless misses in multiprocessors. In Proceeding of the 20th Annual International Symposium on Computer Architecture, pp. 88–97, 1993.

[112] Bradford M. Beckmann and David A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In Proceeding of the 37th international Symposium on Microarchitecture (MICRO-37), Portland, Oregon, pp. 319-330, Dec 2004.

[113] H. Dybdahl and P. Stenström. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In Proceedings of the 13th International Symposium on High-Performance Computer Architecture, pp. 2-12, 2007.

[114] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In Proceedings of the 33rd International Symposium on Computer Architecture, pp. 264-276, 2006.

[115] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In Proceedings of the 32nd International Symposium on Computer Architecture, 2005.

[116] M. Zhang and K. Asanovi´c. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In Proceedings of the 32nd International Symposium on Computer Architecture, pp. 336-345, 2005.

[117] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling cache sharing on chip multiprocessor architectures. In Proceeding of the IEEE International Symposium on Workload Characterization, pp. 160-171, 2006.

[118] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In Proceeding of the 15th International Conference on Parallel architectures and Compilation Techniques, Seattle, pp. 2-12, 2006.

[119] J. D. Collins and D. M. Tullsen. Runtime identification of cache conflict misses: The adaptive miss buffer. ACM Transaction on Computer System, vol. 19, issue 4, pp.413–439, 2001.

[120] G.Memik, G. Reinman, and W. H.Mangione-Smith. Reducing energy and delay using efficient victim caches. In Proceeding of the 2003 International Symposium on Low Power Electronics and Design, Seoul, pp. 262-265, 2003.

[121] Jason Mars Lingjia Tang Mary Lou Soffa. Directly Characterizing Cross Core Interference through Contention Synthesis. In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, HiPEAC, pp. 167-176, January, 2011.

[122] A. Sandberg, D. Ekl¨v, E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11, November, 2010.

[123] N. Topham, A. Gonzalez, and J. Gonzalez. The design and performance of a conflict-avoiding cache. In Proceeding of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 71– 80, 1997.

[124] C. Zhang. Balanced cache: Reducing conflict misses of direct-mapped caches. In Proceeding of the International Symposium on Computer Architecture, Boston, pp. 155-166, 2006.

[125] R. Ricci, S. Barrus, and R. Balasubramonian. Leveraging bloom filters for smartsearch within nuca caches. In Proceedings of the 7th Workshop on Complexity-Effective Design, 2006.

[126] M. Hammoud, S. Cho, and R. Melhem. Dynamic cache clustering for chip multiprocessors. In Proceedings of the International Conference on Supercomputing, pp. 56-67, 2009.

[127] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son. A novel migration-based nuca design for chip multiprocessors. In Proceedings of the International Conference on Supercomputing, pp. 1-12, 2008.

[128] J. Lira, C. Molina, and A. González. Last bank: dealing with address reuse in non-uniform cache architecture for cmps. In Proceedings of the 15th International Euro-Par Conference (Euro-Par), pp. 297-308, 2009.

[129] N. Muralimanohar and R. Balasubramonian. Interconnect design considerations for large nuca caches. In Proceedings of the 34th International Symposium on Computer Architecture, pp. 369-380, 2007.

[130] M. Chaudhuri. Pagenuca: Selected policies for page-grain locality management in large shared chip-multiprocessors. In Proceeding of the 15th International Symposium on High-Performance Computer Architecture, pp. 227-238, 2009.

[131] J. Merino, V. Puente, and J. A. Gregorio. Sp-nuca: A cost effective dynamic non-uniform cache architecture. ACM SIGARCH Computer Architecture News, vol. 36, issue 2, pp. 64–71, May 2008.

[132] M. Hammoud, S. Cho, and R.Melhem. Acm: An efficient approach for managing shared caches in chip multiprocessors. In Proceedings of the 4th International Conference on High Performance and Embedded Architectures, pp. 355-372, 2009.

[133] A. Pesterev, N. Zeldovich, and R. Morris. Locating cache performance bottlenecks using data profiling. In Proceedings of the 5th European conference on Computer systems, EuroSys '10: Pages 335-348, April, 2010.

[134] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In Proceeding of the 36th International Symposium on Microarchitecture, pp. 55-56, 2003.

[135] Liu C, Sivasubramaniam A, Kandemir M, Irwin MJ. Enhancing L2 organization for CMPs with a center cell. In Proceedings of the 20th International Parallel and Distributed Processing Symposium, pp.10-16, 2006.

[136] Wenisch TF, Wunderlich RE, Ferdman M, Ailamaki A, Falsafi B, Hoe JC. Simflex: Statistical sampling of computer system simulation. In IEEE Micro, vol. 26, issue 4, pp.18–31, 2006.

[137] S. Akioka, F. Li, K. Malkowski, P. Raghavan, M. Kandemir, and M. J. Irwin. Ring data location prediction scheme for non-uniform cache architectures. In Proceedings of the International Conference on Computer Design, pp. 693-698, 2008.

[138] Suh GE, Rudolph L, Devadas S. Dynamic Cache Partitioning for CMP/SMT Systems. Journal of Supercomputing, pp. 7–26, 2004.

[139] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. In Proceedings of the 36th International Symposium on Computer Architecture, pp. 3-14, 2009.

[140] A.V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In Proceedings of the 13th international conference on Supercomputing, pp. 145-154. ACM, 1999.

[141] J.B. Rothman., and A.J. Smith. Sector cache design and performance. In Proceedings of the 8th Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000, pp. 124-133, 2000.

[142] J.B. Rothman, and A.J. Smith. "Minerva: An adaptive subblock coherence protocol for improved smp performance." High Performance Computing. Springer Berlin Heidelberg, pp. 64-77, 2002.

[143] H. Kim, and P.V. Gratz. "Leveraging Unused Cache Block Words to Reduce Power in CMP Interconnect." Computer Architecture Letters, vol. 9, issue 1, pp. 33-36, 2010.

[144] Pujara, and A. Aggarwal. "Cache noise prediction." Computers, IEEE Transactions, pp. 1372-1386, (2008).

[145] F.C. Chen, S.H. Yang, B. Falsafi, and A. Moshovos. "Accurate and complexity-effective spatial pattern prediction." In Proceedings of the Software IEE, pp. 276-287, 2004.

[146] D.H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez. The dynamic granularity memory system. In ACM SIGARCH Computer Architecture News, vol. 40, issue 3, pp. 548-559, 2012.

[147] A. Seznec. "Decoupled sectored caches: conciliating low tag implementation cost." In ACM SIGARCH Computer Architecture News, vol. 22, issue 2, pp. 384-393, 1994.

[148] C. Dubnicki, and T. J. LeBlanc. "Adjustable block size coherent caches." In ACM SIGARCH Computer Architecture News, vol. 20, issue 2, pp. 170-180, 1992.

[149] S. Carr, K.S. McKinley, and C-W Tseng. “Compiler optimizations for improving data locality”. vol. 28, issue 5, 1994.

[150] M.E. Wolf., and M.S. Lam. A data locality optimizing algorithm. ACM Sigplan Notices, vol. 26, issue 6, pp. 30-44, 1991.

[151] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In Proceedings of the 13th International Symposium of High-Performance Computer Architecture, pp. 250-259, 2007.

[152] Sparsh Mittal, Yanan Cao, and Zhao Zhang, MASTER: A Multi-core Cache Energy-Saving Technique Using Dynamic Cache Reconfiguration, In IEEE Transactions on very large scale integration (VLSI) systems, Vol. 22, No. 8, pp.1653 -1665, August 2014.

LIST OF PUBLICATIONS

JOURNAL PAPERS

1. Nitin Chaturvedi, S Gurunaryanan "An Efficient block pinning for multi-core architectures", Journal of Microprocessor and Microsystems", Elsevier, ISSN 0141-9331, 39(3), pp.181-188, 2015.
2. Nitin Chaturvedi, Arun Subramanian, S Gururnarayanan, "An Efficient data access policy for shared last Level Cache", WSEAS transaction on computers, ISSN 2224-2872, 14(5), pp.38-48, 2015.
3. Nitin Chaturvedi, Arun Subramanian, S Gururnarayanan, "Selective cache line replication scheme in Shared Last Level Cache", Procedia of Computer Science, Elsevier, ISSN 1877-0509, pp.1095-1107, 2015.
4. Nitin Chaturvedi, S Gururnarayanan, "Adaptive Block Pinning: A Novel Shared Cache Partitioning Techniques for CMP", European Journal of Scientific Research, ISSN 1450-216X, 124(1), pp. 80-93, 2014.
5. Nitin Chaturvedi, S Gururnarayanan "Study of Various Factors Affecting performance of Multi-core architectures" in International Journal of Distributed and Parallel Systems (IJDPS), ISSN 0976-9757, 1(2), pp. 37-45, 2013.
6. Nitin Chaturvedi, Jithin Thomas, S Gururnarayanan "Adaptive Zone-Aware Multi-bank on Chip last level L2 cache Partitioning for Chip Multiprocessors" in the International journal of Computer Applications, , ISSN 0123-4560, 6(9), pp. 19-23, 2010.
7. Nitin Chaturvedi, S Gurunaryanan "An Adaptive Migration-Replication Scheme (AMR) for Shared Cache in Chip Multiprocessors " manuscript submitted in Journal of Parallel Computing" (Elsevier, initial submission 30 October 2014, revision communicated)
8. Nitin Chaturvedi, S Gurunaryanan "An A Locality-Aware Variable Granularity Cache Architecture " revision submitted in Electronics Letter-IET on January 2015(communicated)

CONFERENCE PAPERS

9. Nitin Chaturvedi, Rakesh Kumar, TSB Sudarshan “Adaptive Block Pinning for Multi-core Architectures” in proceedings of International Conference on High Performance Computing **(HiPC)**, Dec-2008

10. Nitin Chaturvedi, Pradeep Harinderan, S Gururnarayanan “A Novel shared L2 NUCA cache partitioning scheme for Multi-core Architectures” in proceedings of International Conference on Emerging Trends in Engineering (ICETE), pp. 183-188, Feb-2010.

11. Nitin Chaturvedi, Prashant Gupta, S Gurunarayanan “Efficient Cache Migration Policy for Chip Multi-Processors” 2011 IEEE International Conference on Computational Intelligence and Computing Research, ICCIC-11, pp. 102-107,2011 **(IEEE-Explore).**

12. Nitin Chaturvedi, S Gurunarayanan “An Adaptive Block Pinning Cache for Reducing Network Traffic in Multi-Core Architectures” 2013 IEEE International Conference on Computational Intelligence and Communication Network, ICCN-2013, pp. 446-450, September 2013. **(IEEE-Explore)**

13. Nitin Chaturvedi, S Gurunarayanan “An Adaptive Cache Coherence Protocol with adaptive Cache for Multi-core Architectures” in proceedings of International Conference on Advanced Electronic Systems, ICAES-2013, pp. 197-201, September 2013, (**IEEE-Explore**)

BRIEF BIOGRAPHY OF CANDIDATE

Nitin Chaturvedi is a Lecturer in the Department of Electrical & Electronics Engineering in Birla Institute of Technology and Science, Pilani since August 2008. Prior to this he worked as Assistant Lecturer in EEE/IU. His interest includes, Energy efficient storage systems, CMOS VLSI Design, Computer Architectures. He obtained his Master of Science (Electronics) from Devi Ahilya University, Indore (M.P) and Master of Technology from University Centre for Instrumentation and Microelectronics (UCIM) from Panjab University, Chandigarh.

BRIEF BIOGRAPHY OF SUPERVISOR

Dr. S. Gurunarayanan is Professor in the Department of Electrical and Electronics Engineering and he is Dean of Work Integrated Learning Program Division (WILPD) in Birla Institute of Technology and Science, Pilani. He obtained Masters in Science (Physics) from Alagappa University, Karaikudi, Masters in Engineering (Systems & Information), from Birla Institute of Technology and Science, Pilani, and Ph.D. (Electronics) from Birla Institute of Technology and Science, Pilani in 1987, 1990 and 2000 respectively. He has several publications in National and International Journals. His research interests are Digital Design and Computer Architecture, VLSI Design, Embedded Systems.