BOOSTING PERFORMANCE OF DIRECTORY-BASED CACHE COHERENCE PROTOCOLS BY DETECTING PRIVATE MEMORY BLOCKS AT SUBPAGE GRANULARITY AND USING A LOW COST ON-CHIP PAGE TABLE

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Mohammad Reza Soltaniyeh

July, 2015

Boosting Performance of Directory-Based Cache Coherence Protocols by Detecting Private Memory Blocks at Subpage Granularity and Using a Low Cost On-Chip Page Table

By Mohammad Reza Soltaniyeh

July, 2015

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Özcan Öztürk(Advisor)

Assoc. Prof. Dr. Çiğdem Gündüz Demir

Assoc. Prof. Dr. Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

BOOSTING PERFORMANCE OF DIRECTORY-BASED CACHE COHERENCE PROTOCOLS BY DETECTING PRIVATE MEMORY BLOCKS AT SUBPAGE GRANULARITY AND USING A LOW COST ON-CHIP PAGE TABLE

Mohammad Reza Soltaniyeh
M.S. in Computer Engineering
Advisor: Assoc. Prof. Dr. Özcan Öztürk
July, 2015

Chip multiprocessors (CMPs) require effective cache coherence protocols as well as fast virtual-to-physical address translation mechanisms for high performance. Directory-based cache coherence protocols are the state-of-the-art approaches in many-core CMPs to keep the data blocks coherent at the last level private caches. However, the area overhead and high associativity requirement of the directory structures may not scale well with increasingly higher number of cores.

As shown in some prior studies, a significant percentage of data blocks are accessed by only one core, therefore, it is not necessary to keep track of these in the directory structure. In this thesis, we have two major contributions. First, we showed that compared to the classification of cache blocks at page granularity as done in some previous studies, data block classification at subpage level helps to detect considerably more private data blocks. Consequently, it reduces the percentage of blocks required to be tracked in the directory significantly compared to similar page level classification approaches. This, in turn, enables smaller directory caches with lower associativity to be used in CMPs without hurting performance, thereby helping the directory structure to scale gracefully with the increasing number of cores. Memory block classification at subpage level, however, may increase the frequency of the operating system's involvement in updating the maintenance bits belonging to subpages stored in page table entries, nullifying some portion of performance benefits of subpage level data classification. To overcome this, we propose as a second contribution, the distributed on-chip page table. The proposed on-chip page table stores recently accessed

pages in the system.

Our simulation results show that, our approach reduces the number of evictions in directory caches by 58%, on the average. Moreover, system performance is improved further by avoiding 84% of the references to OS page table through the on-chip page table.

Keywords: Cache Coherence Protocol, Directory Cache, Many-core Architecture, Virtual-to-Physical Page Translation, Page Table.

ÖZET

ÖZEL BLOKLARIN ALT SAYFA SEVİYESİNDE TESPİT EDİLMESİ VE DÜŞÜK MALİYETLİ YONGA ÜZERİ SAYFA TABLO KULLANILMASIYLA DİZİN TEMELLİ ÖNBELLEK TUTARLIĞI VERİMLİLİĞİNİN ARTIRILMASI

Mohammad Reza Soltaniyeh
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Doç. Dr. Özcan Öztürk
Temmuz, 2015

Çok çekirdekli işlemcilerde (CMP) yüksek performans sağlamak için etkili önbellek tutarlılık protokolleri ve yanı sıra hızlı sanal-fiziksel adres çeviri mekanizmaları gerekir. Dizin (Directory) temelli önbellek tutarlılık protokolleri çok çekirdekli işlemcilerde, veri bloklarının son seviye özel önbelleklerde tutarlı bir şekilde bulunmasını sağlamak amacıyla yaygın bir şekilde kullanılan bir yaklaşımdır. Ancak, dizin yapılarının büyük fiziksel alan işgal etmeleri ve önbellek ilişkilendirmesinin yüksek olması sebebiyle, çekirdek sayısı çoğaldıkça ölçeklenebilirlik derecesi düşebilir.

Daha önceki çalışmalarda gösterildiği gibi, veri bloklarının önemli bir yüzdesi sadece tek bir çekirdek tarafından erişilir. Bu nedenle, dizin yapısında bu veri bloklarını takip etmek gerekli değildir. Bu tez, iki büyük katkıyı sunmaktadır: ilk olarak, daha önceki çalışmalarda önerilmiş olan sayfa düzeyini göze alarak sınıflandırmaya göre, alt sayfa düzeyinde veri bloklarını sıflandırmanın önemli bir ölçüde daha çok özel veri bloklarının tespit edilmesine yardımcı olabileceğini gösterdik. Sonuç olarak, bu yaklaşım, benzer sayfa düzeyinde sınıflandırma yaklaşımlarına göre, dizinde takip edilmesi gereken blokların yüzdesini önemli bir ölçüde düşürür. Bu da, olabildiğince çok çekirdekli işlemcilerde performansa zarar vermeden daha küçük ve daha az ilişkilerdirmeli olan önbellek dizinlerinin kullanılmasını mümkün kılıp, böylece dizin yapısının çekirdek sayısının çoğalmasıyla beraber ölçeklenmesine yardım eder. Ancak alt sayfa düzeyinde bellek bloğu sınıflandırma, işletim sisteminin sayfa tablosu girdilerinde saklanan alt sayfalara ait bakım bitlerinin güncelleme frekansının yükselmesine neden olabilir. Bu

yüzden alt sayfa düzeyinde veri sınıflandırma performans avantajlarının bir kısmı boşa çıkabilir. Bunun üstesinden gelmek için, ikinci bir katkı olarak, bu tezde dağıtımlı yonga üstü sayfa tablosu kavramı önerilmektedir. Önerilen yonga üstü sayfa tablosu sistemde en son erişilmiş olan sayfaları saklamaktadır.

Simülasyon sonuçlarımıza göre, önerdiğimiz yöntem ortalama olarak dizin belleklerinin tahliye sayısı oranını 58% azaltmaktadır. Ayrıca, yonga üstü sayfa tablosu işletim sistemi sayfa tablosu erişimini 84% azaltıp sistem performansında artışa yardımcı olmaktadır.

Anahtar sözcükler: Önbellek Tutarlılık Protokolü, Dizin Önbelleği, Çok Çekirdekli Mimari, Sanal-Fiziksel Sayfa Dönüşümü, Sayfa Tablosu.

Acknowledgement

Bilkent University provided me with the opportunities that opened the doors in my life, both personally and professionally. I thank my adviser Dr. Özcan Öztürk for supporting my pursuit of this M.Sc. He is truly the one that everybody dreams of having as an adviser. I will never forget his endless kindness to me. I also thank Dr. İsmail Kadayıf for supporting me throughout this research and trusting me to be part of the research which was funded by the Science And Technological Research Council of Turkey (TUBITAK). I also thank the jury members, Dr. Çiğdem Gündüz Demir and Dr. Süleyman Tosun for their helpful feedbacks and their precious comments.

Dr.Öztürk research group was a rewarding experience because of all the friends I met there. I thank Naveed, Şerif, and Erdem that helped me a lot during these two years. I also thank my Iranian friends in the group; Hamzeh, Salman and Arghavan for their support.

Finally, I want to thank my whole family, specially, my mother and father. I know it has been so hard for them to spend months far from their darling child, but, they never complained about it! I know, I'd never had what I have now, without any of their prays for me.

Contents

List of Figures

Chapter 1

Introduction

While the technology facilitated the integration of more transistors on a single chip as predicted by Moore's Law [2], it is not clear how these excessive number of transistors are supposed to increase performance. One pervasive approach for boosting performance with the additional transistor resources is to increase the number of functional units involved in processing. For achieving this, Chip Multiprocessors (CMPs) which enable multiple processing units (cores) on a single chip, have been studied by several works in the recent decade.

A system with many cores inside, however, introduces many complications for the programmers who are working with that system. Particularly, managing data movement and data storage in a system with many cores inside can be very hard for the system. In order to reduce the complexity, shared-memory architectures are introduced which provide a single view of memory shared by all the processors in the chip. This relieves programmers and the system from most of the burdens of managing data accesses across different threads. Although, shared-memory paradigm is advantageous from a software point of view, it introduces additional complications at the hardware level.

One of the complication is providing a consistent view of memory with different

levels of caches, that is keeping the whole on-chip memory system coherent. Although cache coherence is abstracted from the software, it is still a major factor for consistency and correctness of a CMP with shared-memory paradigm. As such, cache coherence protocols were introduced to provide a coherent cache system for chip multiprocessor systems. Cache coherence management schemes not only affect the correctness of a shared-memory CMP, but also impact the system performance. Potentially, ineffective cache coherence management can nullify all the benefits we gained by running applications in parallel with CMPs. This is the main reason why cache coherence has been studied by many researches in the literature.

Cache coherence management is facing new issues as technology continues to scale. Previous approaches for maintaining coherence do not scale well with higher number of cores. Broadcast-based snoopy protocols commonly used in bus based CMPs, can no longer be used as an efficient way of managing cache coherence in many-core architectures. Excessive bandwidth requirements of these protocols, as well as the need for an interconnection network that provides ordering globally, are the two main limitations that broadcast-based cache coherence protocols faced.

On the other hand, directory-based protocols were introduced to eliminate broadcasts by storing the sharing information in a structure named *directory*. The directory requires additional storage area to record the list of sharers together with the coherence states for each cache block. Using directory effectively prevents exchanging huge number of snooping messages required by snoopy-based cache coherence protocols by allowing the coherence messages sent and received only by the cores involved. The directory also serve as the ordering point for requests, and thereby, enables using more scalable interconnect designs than bus-based interconnects.

Although directory-based cache coherence protocols are considered to be the best solution for a scalable scheme for managing cache coherence in a system with many cores, the directory can become a limiting factor in implementing a directory-based coherence scheme for large scale CMPs. This is because of the fact that the number of blocks as well as the size of each block in the directory,

increases linearly with the number of cores and the cache capacity. To put this problem into perspective, a typical full-map directory records a bit vector for each core in the chip. Furthermore, the cache requirements keep continuously increasing. The state-of-the-art architectures no longer have a single layered caches, instead, they utilize multiple layers. In such a setting, upper layers have private caches, whereas lower levels have shared caches. As a result of this, the number of blocks needed to be kept coherent is also growing dramatically. These two aforementioned points make directory design in a many-core CMP very tricky as the directory itself may add significant area and energy cost, thereby affects the performance of the whole system.

Different directory organizations have been suggested to reduce the overhead and improve the scalability in coherence protocols relying on directories. The details of the more commonly used approaches in directory-based cache coherence protocols are discussed in detail in Chapter 2 .

This work intends to reduce the overhead of coherence management by proposing a runtime data classification. The aim of the proposed data classification is to find those data blocks that do not need coherence operations normally applied for all the cache blocks in state-of-the-art cache coherence management schemes. The effectiveness of a detection mechanism to override cache coherence relies on two factors. First, it should detect a considerably large quantity of blocks which possibly can be discarded for coherence management. Otherwise, the overhead imposed by the detection mechanism would hide the expected benefits. Second, the detection mechanism itself should not be very costly in terms of area, design complexity, and power consumption. Otherwise, it will not provide the performance benefits. In other words, the granularity in which we are detecting blocks that do not need coherence, should be intelligently decided to provide us an effective coherence management scheme.

Another aspect of a high performance memory system is the support for fast virtual to physical memory address translation. Translation Look-aside Buffers (TLBs) are the key elements in supporting for the virtual memory management. Approaches for designing TLBs are becoming more and more important as they

commonly exist in the critical path of memory accesses. The aim of studies in this area is mainly lowering the address translation latency as much as possible and keeping it off the critical path of memory accesses.

Several techniques have been proposed to reduce virtual address translation for the uniprocessor architectures [3] [4] [5] [6]. There are also recent studies on TLB organization in the context of CMPs [7] [8]. A pervasive approach for organizing the TLB in many-core architectures is the use of per-core private TLB. However, it is shown in [9] that such an organization may lead to poor performance and still keep the TLB in critical path of memory accesses. The proposed approaches in this domain can be classified in to two categories. The first one suggests a shared last level TLB [8] [10] to improve sharing capacity among the cores in the system. In contrast, the approaches in the second category, try to increase sharing capabilities between the cores by enabling a cooperation between private TLBs [7]. In other words, each individual core tries to borrow capacity from the private TLBs of other cores in the system before performing the translation with a relatively higher cost in Operating System (OS). As will be discussed later, both of these techniques have its own shortcomings. Using a shared TLB introduces higher access latency, requires a high bandwidth interconnection network, and of course, needs a higher associativity, leading to higher power consumption. On the other hand, the approaches in the second category need inquiring other TLBs for each missed page translation that happens in each core which can introduce both design complexity and also higher network traffic.

In this thesis, we propose a technique to decrease the cost of virtual memory address translation in CMPs which does not suffer from the shortcomings discussed earlier.

1.1 Thesis Contributions

We can summarize the contributions of this dissertation as follows:

- First, we propose a data block classification methodology which works at subpage level and helps to detect considerably more private data blocks. Consequently, it reduces the percentage of blocks required to be tracked in the directory significantly compared to similar page level classification approaches. This, in turn, enables smaller directory caches with lower associativity to be used in CMPs without hurting performance, thereby helping the directory structure to scale gracefully with the increasing number of cores.

- Second, we propose a small distributed table referred to as the on-chip page table, which stores the page table entries for recently accessed pages in the system. This can be implemented as a portion of the directory controller. Upon a TLB miss, the operating system gets involved in address translation only when the translation is not found in the on-chip page table. It also helps to negate performance degradation that might have occurred due to the increase in the frequency of the operating system involvement in our subpage granularity block classification.

1.2 Thesis Organization

Chapter 2 provides the background and discusses the related studies in more detail. Chapter 3 lists the motivations for this dissertation, whereas Chapter 4 describes our approach to design an effective directory-based cache coherence scheme for many-core CMPs. We describe the details of both contributions with the help of examples and figures. Chapter 5 gives the details of our system configuration, tools, system parameters and benchmarks used. In Chapter 6, we demonstrate the performance improvement we can get from our proposed ideas. Finally, Chapter 7 concludes the dissertation and discusses the possible directions for future research.

Chapter 2

Background

This chapter gives an overview of cache coherence problem and present a summary of the important techniques proposed in this subject. Since there are many efforts in the field, we only focus on the most relevant topics to our proposal. Section 2.1 discusses the role of cache coherence problem in providing memory consistency for multiprocessors. Section 2.2 explains cache coherence problem with the help of an example. In Section 2.3, we summarize the efforts made to efficiently address cache coherence problem for chip multiprocessor systems. Section 2.4, discusses the directory organization in directory-based cache coherence protocols in more details. In Section 2.5, we classify existing mechanisms that tried to address the overhead of managing cache coherence in chip multiprocessors. Finally, we give an overview of most important studies on fast virtual memory management in Section 2.6.

2.1 Memory Consistency Problem

In Von Neumann architecture, instructions are executed in the order specified by the programmer or compiler. The important feature of the model is that, a program's *load* instruction returns the last modified value of a block (as a result

of the last *store* operation that corresponds to the memory block). This provides a simple semantic for uniprocessor architectures.

Multithreaded programs, however, enforce new complexities to the aforementioned model, since the most recent value for a block might be the result of a store on a different core. Therefore, *memory consistency models* [11] have been suggested to identify how a core running a thread should behave for memory accesses from other cores in the system.

The most straightforward model to address the ordering problem of the operations in a multithreaded program is to execute operations inside each single core in the order specified by the program. This is named as *sequential consistency model* [12]. In contrast, there are relaxed consistency models [11] that provide a more flexible model with space for different optimizations. While, relaxed model can improve the performance by allowing some of the memory operations to be executed before they are observed by the other cores, properly synchronizing the multithreaded program is still needed to be applied either by the programmer or by the compiler [13].

Regardless of which consistency model is being used, introducing cache memories to the system affects the implemented consistency model. This problem is known as *cache coherence problem* which is part of the memory consistency model. We will elucidate this problem in next section.

2.2 Cache Coherence Problem

Cache system plays an important role in performance of Chip Multiprocessors (CMPs) by filling the speed gap between processor and memory. However, implementing cache hierarchy introduces some difficulties in case of a multiprocessor system. Following example can clarify the cause of the *cache coherence problem*. Suppose two cores in a system, C1 and C2, load the same memory block (B1) into their respective private caches. When C1 updates B1 with a new value. The

caches in the system (C1 and C2 private caches) can become inconsistent. However, if C2 never loads B1 again, there would be no such problem. In order to support a shared-memory programming paradigm for a multiprocessor system, we need to ensure coherence between all levels of caches located in all the cores in the system. The mechanism applied to maintain such a coherence between caches exist in the system is known as *cache coherence mechanism*.

We can call a system, cache coherent; if it keeps a valid order of reads and writes to a memory location. In other words, each read operation should return the value written by the last write to that location. It can be either on the same core, or any other core in the system.

Although, cache coherency is a vital component of a multiprocessor consistency model discussed earlier, it is not enough to ensure consistency. Nonetheless, it is coherence protocol's job to indicate completion of load and store operations which is required to be known by consistency model to enforce ordering requirements.

A common approach to maintain all blocks located in a cache system coherent is by assigning each block in the cache an access permission value. For instance, in a single writer-multiple reader model, only one of the cores can acquire a cache block with write permission. Other cores store the same block in their caches with only read permission. These permission values can be considered as a coherence state stored for each block in the cache. The *coherence protocol* ensures a right state would be assigned to each cache block with respect to the state of each block currently present in all the cores in the system. For example, a core may have an exclusive write permission to a certain block if all other copies of that block in other caches are all in invalid state.

Different cache coherence protocols may define different set of coherence states. A very basic and minimum set of states for a cache coherence protocol to include, consists of three basic states named as M, S, and I. State (M) denotes a single processor holding a block with a write permission. State (S) allow multiple cores to keep the same block simultaneously, whereas state (I) means that the cores do not own the block.

Although, above set of states might be enough to ensure cache coherence, for the sake of optimization, state-of-the-art cache coherence protocols usually use more states than the basic three states to ensure coherence in caches. Table 2.1 provides a list of common states utilized in cache coherence protocols with their definition. Most of the existing cache coherence protocols use a subset of these states.

A core in the system may issue different type of requests to obtain a block with a certain state (permission). For instance, a core can issue a GETS (GET Shared) to get data only with a read permission. Or, it may need to acquire a block with a write permission with GETM (GET Modified) request. Cache coherence protocol is responsible for managing all the cache states and responding to the requests coming from the cores. Cache coherence protocol enforces sending and receiving requests between the cores to keep the block states in a proper way. These messages are known as coherence messages. This implies a traffic overhead (known as *coherence traffic*) between the cache controllers in the system.

Table 2.1: Cache coherence states

States	Permission	Definition
Modified (M)	read,write	other caches in I or NP
Owned (O)	read	other caches in S, I, or NP
Exclusive (E)	read,write	other caches in I or NP
Shared (S)	read	no other cache in M or E
Invalid (I)	none	none
Not Present (NP)	none	none

2.3 Cache Coherence Techniques

As discussed previously, cache coherence protocol is needed to maintain caches in the system coherent. There are two main types of cache coherence protocols in the literature; *Snoopy-based* cache coherence protocols and *directory-based* cache coherence protocols. We will discuss these in detail in the following section.

2.3.1 Snoopy-based Cache Coherence

In a snoopy-based cache coherence protocol, to ensure that there is only a single copy of a cache block with write permission, each core broadcasts coherence messages to all other cores (regardless of the state of the block in those cores). Each core snoops to see if it has a copy of that cache block and responds accordingly. Obviously, this approach needs to keep track of the order of the requests. Thus, it is usually used in a bus-based system in which ordering is ensured by the network. Implementing snoopy-based cache coherence protocol on a network that does not preserve ordering, introduces so many difficulties. There are works that use snoopy based protocol on ring topology [14] and other arbitrary topologies [15] [16] [17].

Snoopy-based protocols are pervasive approaches in a system with few number of cores in a chip. However, bandwidth requirements, and limitations on the topologies that support snoopy-based cache coherence protocols, cause some serious scalability problems, especially with higher core counts. As a result, *directory-based* cache coherence protocols were proposed to address these scalability problems.

2.3.2 Directory-based Cache Coherence

To address high bandwidth requirements of broadcast-based cache coherence protocols (snoopy) and also enable coherence on interconnect networks that do not necessarily preserve ordering, the directory-based mechanism was first suggested by Censier [18] and Tang [19] and soon, commercial machines [20] [21] used directory-based protocols to maintain cache coherence.

A directory-based cache coherence protocol contains a directory that stores the sharing status of a certain block. Typically, a directory consists of a list of sharers for each block, in which, the current owner is also identified. There are different proposals how this information can be kept efficiently in the directory [22] [23].

In a directory-based cache coherence protocol, each core refers to the directory to inquire the sharing status, as well as the sharer information, before sending required messages to other cores. The information kept in the directory enables the core to unicast the right message to the cores really have the corresponding data block. So, it avoids high bandwidth requirements forced by snoopy-based protocols. Moreover, with a level of indirection, the directory-based cache coherence protocols are able to work on top of any interconnect network without introducing any additional complexity [21] [24].

Although, the directory-based cache coherence protocols are state-of-the-art approaches to maintain cache coherence in many-core chip multiprocessors, directory organization can still become a bottleneck when scaled. In the next section. we review the important proposals on organization of directories.

2.4 Directory Organization

In directory organization for coherence, for each data block there must be a single entry in the directory. Directory structure have been studied from many angles. while some studies tried to find how the sharing information for each block (directory width) can be squeezed to enable a more scalable directory organization [22] [23] [25], some others aimed at decreasing the size of the directory structure by only keeping track of a subset of all memory blocks available [26] [27]. Moreover, the location of the directory and how directory accesses should be managed have also been the topic for a set of studies [21] [28] [29]. In this section, we summarize the most important proposals that particularly studied directory organization.

Some proposals tried to reduce the size of each entry in the directory by compressing the sharing information instead of a full map approach. Some examples of a compressed sharing codes are tristate [23], gray-tristate [25] and binary tree with subtrees [22]. A different approach is the coarse vector method which is based on keeping a bit of sharing code for a set of K processors. In other studies, authors suggested a limited number of pointers for each single entry, that do not

cover all the sharers [30] [31]. This implies that some cases must be handled by broadcasting messages or eliminating one of the existing copies.

Reducing the height of the directory is as important as previous proposals on decreasing the width of the directory. Thus, many studies studied the method to decrease the total number of entries in the directory. In [32], authors do this by combining several entries into a single one. One other approach does the reduction by organizing the directory structure as sparse directory [33]. Some proposals limited the directory entries to the blocks cached in the private caches [34]. These techniques result in extra coherence messages needed to be exchanged.

For a distributed shared-memory chip multiprocessor, a common approach is having multiple duplicate tags. For instance, in Everest [35], each distributed directory keeps the state information for the blocks belonging to the local home but cached in the remote nodes as well. Using this kind of directory organization, the number of entries in the distributed directories grows linearly with the number of cores in the system. Other techniques studied interleaving the entries into the distributed directories in order to reduce the size of the distributed directory [36]. Unfortunately, this approach also suffers from extra latency due to accessing the full list of pointers required for interleaving.

The directory structure can be centralized as it is in Piranha [37], or distributed like Sun UltraSPARC T2 architecture [**?**]. In a centralized directory structure, all cache misses must access the same directory which causes a bottleneck for many-core CMPs. On the contrary, a distributed directory is a more scalable solution for many-core CMPs, where each directory is responsible for keeping track of memory blocks in its home tile. A distributed directory is considered scalable if the size of each directory slice does not necessarily vary with the number of tiles in the system. A popular way of organizing distributed directory in tiled CMPs is the use of directory caches [29].

A *directory cache* provides faster accesses to a subset of a complete directory. Directories keep track of coherent states and exhibit the same locality as data and

instruction caches. Introducing directory caches does not influence the cache coherence protocol functionality. In a state-of-the-art many-core architecture where the cost of accesses to other cores is not very high, the latency of a costly off-chip directory access can become the bottleneck. Therefore, for those architectures, there is a motivation for a fast directory access via on-chip directory cache to avoid off-chip accesses. The different strategies for designing directory caches have been summarized in the following section.

2.4.1 Directory Cache Organization

One way of organizing the directory is keeping a complete directory in DRAM, and then use a separate directory cache to reduce the average access latency. Directory caches in this approach is decoupled from the Last Level Cache (LLC), therefore, it is possible to experience a costly DRAM access due to a miss in the directory cache, even if the block is available in the LLC. Further, any directory replacement must write back to DRAM, causing high latency and power overheads.

A more efficient way of organizing directory cache is based on the fact that we only need to keep track of blocks that are being stored in one of the caches on the chip. This type of directories are referred to as *inclusive directory cache*. An inclusive directory only caches blocks that are located somewhere on the chip. This implies that a miss in an inclusive directory puts the block in state I. So, there is no need for a complete directory in DRAM to back the directory cache.

The simplest directory cache design relies on LLC inclusion. Cache inclusion implies that if a block exists in upper-level caches, it also must be present in lower-level caches. In an inclusive directory cache may benefit from this property by keeping the coherence states for each block in the same place as the data kept in the LLC. In the case of a miss in the LLC, the directory controllers know that the requested block is not cached in any other core on the chip.

Adding extra bits to each block in LLC can lead to a non-trivial overhead depending on the size of the system in terms of number of cores and the format in which directory states are presented. Moreover, LLC inclusion has serious drawbacks. Maintaining inclusion for a shared level cache, requires sending special requests to invalidate blocks from the private caches when a block is replaced in the LLC. More importantly, LLC inclusion needs to keep redundant copies of cache blocks that are in upper-level caches. The situation may be more dramatic in many-core CMPs where the collective capacity of the upper-level caches may be a significant portion of the capacity of the LLC.

An inclusive directory cache design which is not supported by an inclusion between different levels of caches, must contain directory entries for the union of blocks in all the private caches. This is because, a block in the LLC but not in any private caches must be in state I. This enforces directory to cache duplicate copies of the tags in all private caches. It is a more flexible design in the sense that the directory cache does not rely on LLC inclusion. However, it suffers from extra storage costs needed for caching duplicate tags.

The inclusive directory caches introduces other significant implementation costs as well. They require a highly associative directory cache. For example, consider a directory cache for a chip with C cores, each of which has a K-way set-associative L1 cache. For this specific example, directory cache must be C*K-way associative to hold L1 cache tags. To put it into perspective, for this directory organization to work, we need a directory cache where its associativity grows linearly with the core counts. Therefore, this approach suffers from high power consumption and high design complexity due to high associativity.

To overcome the scalability problem of the previously discussed directory cache organization, we must limit the associativity to a certain level. So using a lower associativity in directory caches rather than using upper case associativity level (C*K), forces some additional overheads. For example, in the case of full directory, we need to evict an entry before adding the new entry to directory cache. Any eviction in the directory cache requires invalidation messages to be sent to all the caches that hold the evicted block in a valid state.

With the assist of invalidation messages (*recall* requests), we can overcome the need for a high associative directory cache. But, if the number of evictions in the directory cache passes a certain rate, then it may lead to poor performance. To avoid intolerable *recall* requests and get reasonable performance, we should keep the directory evictions of as low as possible. This is one of the main contributions of this thesis.

2.5 Low-Overhead Cache Coherence

Several techniques have been proposed to reduce the overhead of managing coherence specially for large scale CMPs. Most of the optimizations proposed by these studies are based on a some sort of data classification. The suggested classification mechanisms, however are different from various points of view. First, the level in which the classification performed, varies in different studies. Some of the works do the classification at compiler level, other utilize Operating System (OS) to classify the data, and some introduce extra hardware components to classify blocks. Second, the granularity of the classification is also different. Some classify data in a fine-grained block granularity. Whereas, others use a coarse-grained classification methods to avoid the possible overhead of a fine granularity.

Apart from the differences exist in the classification techniques, different studies utilized the classified data for various reasons. In this section, we will review the important proposals on each of the aforementioned topics.

Some prior studies exploited private/shared data detection to enable high performance for many-core architecture by mitigating the overhead of managing coherence. *Private data* refers to the data that is only accessed by a single core. While, *shared data*, is the data that is accessed by more than one core in the system. The classification can be performed by either hardware, compiler, or software, or a combination of those.

POPS [38] optimized the coherence protocol by intelligently placing private

and shared data at different levels of caches in Non-Uniform Cache Architecture (NUCA). They introduce extra *hardware* to decouple the private and shared data which enables utilizing LLC capacity more effectively. They also change the coherence protocol actions to exploit the observed sharing patterns.

In SPTAL [39], authors use a tagless directory [40] together with bloom filters to summarize the tags in a cache set. They observed that majority of bloom filters replicate the same sharing pattern, while suggest to decouple the sharing pattern and eliminate the redundant copies of these sharing patterns. SPTAL can work with both inclusive and non-inclusive shared caches and provides up to 34 % storage savings over other tagless directory approaches.

Cantin et al. [41], perform a coarse-grained tracking to reduce the unnecessary traffic due to broadcast-based protocols. For doing so, they suggest *Region Coherence Arrays* to identify shared regions and then, filter unnecessary broadcast traffic based on the detected shared regions. In turn, RegionTracker [42] introduces a framework that reduces the storage overhead and improves the precision of the two previous studies.

Furthermore, there are studies that are assisted by compiler or OS to classify data and apply further optimizations. As an example, Li at el. [43] proposes a novel compiler-assisted data classification technique to speculatively detect a class of data termed as *practically private*. They demonstrated that their classification provides efficient solutions to mitigate access latency and coherence overhead in many-core architectures. Another relevant effort by Jin and Cho offer a Software Oriented Shared (SOS) cache management [44]. They classify data accesses to a range of memory locations. Their profile method classifies the memory locations returned by malloc() into several categories such as Owner, Dominant, Partition, Scattered, and etc. They further use this profiling information as a hint to place the blocks of memory in an optimized way in cache tiles.

Compiler has been specially used to extract the communication pattern of application in Sha's work [45]. In this work, they extracted the communication pattern of message passing applications statically to minimize runtime circuit

establishment overhead of an optical circuit switching interconnect. They further utilize this pattern extraction to improve TLB structure for fast virtual to physical address translation [46].

Beside these proposals that utilize compiler to statically classify data, there are some runtime mechanisms which are mostly supported by OS to classify data and improve performance based on the result of the classification. One of the most important proposal in this area is Reactive-NUCA (R-NUCA) [47]. In R-NUCA, data accesses are classified as private, shared and read-only at the page granularity. Every page considered as private by default until a second core accesses the data. In R-NUCA, they utilized the classification results by placing private pages locally to improve access latency while shared pages are cached in S-NUCA style [48].

In a different OS-assisted data classification work, Kim et al [49] employs a complex method to detect shared data as well as their sharing degree to reduce the snoops take place in a token-based cache coherence protocol. Their proposal requires extra hardware support as well as OS modification.

Similarly to R-NUCA, but with different motivation, Cuesta et al [50] presented an efficient directory organization based on an OS-based runtime data classification similar to R-NUCA. They modified TLB entries and page table entries to apply their classification. Their proposed method classifies data in a coarse-grained page granularity. In other words, existence of single block of memory accessed by more than one core is enough to consider the whole page as shared. This may penalize the proposed idea particularly, for those architectures that use larger pages for improving performance.

On the other hand, Zeffer's work [51] employed a combination of hardware and software to support cache coherence. They propose a trap-based architecture (TMA), which detects fine-grained coherence violations in hardware, and causes a trap in case of violation. Software, further, maintains coherence with coherence trap handlers. Again, like Cuesta's work, the TLB and OS are exploited to catch pages that move from a private state to a shared one. TMA requires

extra hardware to speed up the coherence trap handlers. Finally, they propose a straightforward hardware mechanism that implements an inter-node coherence protocol in software.

Fensch et al. [52] propose a coherence protocol that does not need any hardware support. They rather avoid any incoherence in caches by not allowing multiple writable shared copies of pages. Pages are mapped to each processor's cache under control of OS and remote cache accesses enabled by the hardware and are very costly. Their proposal requires release consistency, and introduces unacceptable overhead in hardwired systems.

2.6 Virtual Memory Management Techniques

Different techniques such as various TLB organizations, multilevel TLB organizations, prefetching, and etc, have been suggested for virtual memory management in uniprocessor architectures [3] [4] [6]. Since our focus is on CMPs, we review important studies on virtual memory management for CMPs. The pervasive approach for organizing the TLBs in many-core architectures is using per-core private TLB. However, it is shown in [4] that such an organization may lead to poor performance and still keep the TLB in the critical path of memory accesses. To overcome this, different techniques were suggested, which are broadly classified into two categories. The approaches in the first category suggest a shared last level TLB [8] [10] to improve sharing capacity among the cores in the system. In contrast, the approaches in the second category try to increase sharing capabilities between the cores by enabling a cooperation between the private TLBs [1]. In other words, each individual core tries to borrow capacity from private TLBs of other cores in the system before finding the translation with relatively higher cost in OS. We review the advantages and disadvantages of each of these categories in this section.

In a study by Bhattacharjee et al. [10], authors suggest a shared last level TLB inspired by cache organization in modern CMPs. Their proposed shared last level

TLB improves sharing capacity among different cores and the collective number of TLB misses is reduced in their implementation. However, their shared TLB organization has some disadvantageous. First, shared TLBs would introduce higher access latency when compared to private per core TLBs. This excessive latency adversely affects the latency of every memory access. Second, for connecting the shared TLB with all cores, we require a high bandwidth interconnect. This problem becomes more dramatic with higher number of cores on the chip. Third, a shared TLB organization needs a high associativity to avoid high miss ratio leading to design complexity and higher power consumption. Beside these disadvantages, there are additional issues regarding the placement of the shared TLB in a large-scale system to avoid high wire delays.

By characterizing parallel workloads on CMPs, Bhattacharjee and Martonosi [9] showed significant similarities in TLB miss pattern among multiple cores. Based on their observation, the same authors proposed Inter-Core Cooperative (ICC) TLB prefetchers [8]. They were able to avoid 19% to 90% of data TLB (D-TLB) misses with their proposal. Although, their proposal showed a considerable improvement in the context of parallel multithreaded workloads, however, there are some drawbacks in their proposal. First, as they also reported in their work, the percentage of a bad prefetcher can be as high as 60%. To avoid performance loss due to a bad prefetcher, their prefetching mechanism needs to add extra hardware which can be costly. Second, for generic multiprogrammed workload on CMPs where different applications do not share any address translation among each other, the proposed scheme is not very helpful.

To exploit all the benefits of a private TLB organization and, at the same time, improve the performance of virtual memory management by providing a fast address translation, Srikantaiah and Kandemir proposed *Synergistic TLBs* [7] for CMPs. Their proposal relies on minor hardware changes to increase sharing capacity among the private TLBs by borrowing capacity from private TLBs of other cores. The proposed Synergistic TLB has three characteristics. First, it differs private TLB organization by providing capacity sharing of TLBs. Second, it supports translation migration to maximize TLB capacity utilization. Third,

it enables translation replication to prevent high latency for remote TLB accesses. While the cooprating TLB prefetecher proposal discussed earlier, which was not using capacity sharing, Synergistic TLBs reduce misses by adopting capacity sharing. The Synergistic TLB outperforms ICC prefetching in most of the multithreaded applications. Despite their important contribution to the area of designing high performance TLB organization, their proposal may fail to scale with large core counts. Their approach is based on inquiring other TLBs in the case of a miss in private TLBs. Snooping other TLBs in the system for finding the page translation is not scalable with higher core counts due to traffic overhead and excessive energy consumption required by snooping.

Chapter 3

Motivation

In this chapter, we first discuss the approaches given earlier in Chapter 2 and explain their shortcomings which motivated us to work on the topic. Then, we explain the areas where there is still room for improvement in TLB organizations and in general fast virtual memory management. This chapter, in a way, prepares the ground for our proposal.

3.1 Low-Overhead Cache Coherence

As discussed in Chapter 2, directory-based cache-coherence protocols are the common approach for managing the coherence in systems with many cores in a single chip because of their scalability in power consumption and area compared to traditional broadcast-based protocols. However, the latency and power requirements of today's many-core architectures with their large last level caches (LLCs) brought new challenges. A common approach is to cache a subset of directory entries (directory cache) due to high latency and power overheads with directory accesses. A directory cache must provide an efficient way to keep the copies of data blocks stored in different private caches coherent since its structure can have momentous influence on overall system performance.

One simple way of designing directory, as explained before, is to maintain an inclusive cache. But, such a design have some drawbacks. First, it is very costly to maintain inclusion as it needs interchanging lots of message requests between the cache controllers. Second, inclusion requires maintaining redundant copies of cache blocks that exist in upper-layer caches. This is the motivation for some of the previously proposed approaches (including ours) to investigate techniques that do not rely on upper layer inclusion. Duplicate-tag-based directories are common scheme for doing so in CMPs. Compared to other directory structures, this type of directory caches are more flexible as they do not force any inclusion among the cache levels. However, directories based on duplicate-tag come with overheads as mentioned in Chapter 2. Storage cost for duplicate tags, and more notably, high associativity requirement that grows with the number of cores in the system, are two main overheads. Therefore, this approach suffers from high power consumption and high design complexity due to high associative caches.

In this dissertation, we avoid poor performance of low associative directory caches (as a result of high invalidation counts in directory cache) with our proposed directory organization which will be presented in detail in Chapter 4.

Similar to prior studies exploited private data detection to implement high performance and scalable cache coherence management for many-core architectures [47] [50], we also try to utilize a data classification mechanism motivated by our first observation. This way, we aim to achieve a low-overhead and efficient directory-based cache coherence management scheme.

Observation 1. *The granularity at which we detect private accesses can play a vital role in performance benefits we can obtain.*

We observed that, by inspecting private data at a finer granularity than page granularity, chances for finding private data and further improvements will be considerably higher. Figure 3.1 shows the amount of private accesses detected with a subpage granularity (4 subpages per page) compared to a page granularity approach for ten different multithreaded applications in systems with 4, 8, and 16 cores. The reason for such a huge difference is that existence of a single shared

block within a page is enough to change the page status from private to shared. In our example, the page size is 8KB. The difference would be more dramatic for those architectures that use larger pages for improving performance.

According to Figure 3.1, chances of detecting private accesses in subpage granularity is about two times more than doing so in page granularity. We will discuss later in this thesis how we can perform the detection in subpage level with the assistance of page tables and TLB entries.

3.2 Fast Virtual Address Translation

TLBs are the key components for supporting fast translation from virtual to physical addresses. However, in many cases, they are in the critical path of memory accesses. This is the main motivation for many studies mentioned earlier which focus on the techniques for fast and efficient virtual to physical address translation through TLBs.

A pervasive approach for organizing the TLBs in many-core architectures is per-core private TLBs. However, it is shown [4] that such an organization may lead to poor performance and still keep the TLB in critical path for memory accesses. To overcome this, different techniques were suggested, which are broadly classified into two categories. The approaches in the first category suggest a shared last level TLB [2][4] to improve sharing capacity among the cores in the system. In contrast, the approaches in the second category try to increase sharing capabilities between the cores by enabling a cooperation between the private TLBs [1]. In other words, each individual core tries to borrow capacity from private TLBs of other cores in the system before finding the translation with relatively higher cost in the OS.

The approaches using a shared TLB introduce higher access latency, require interconnection network with higher bandwidth, and of course, need a higher associativity, leading to higher power consumption. On the other hand, the approaches

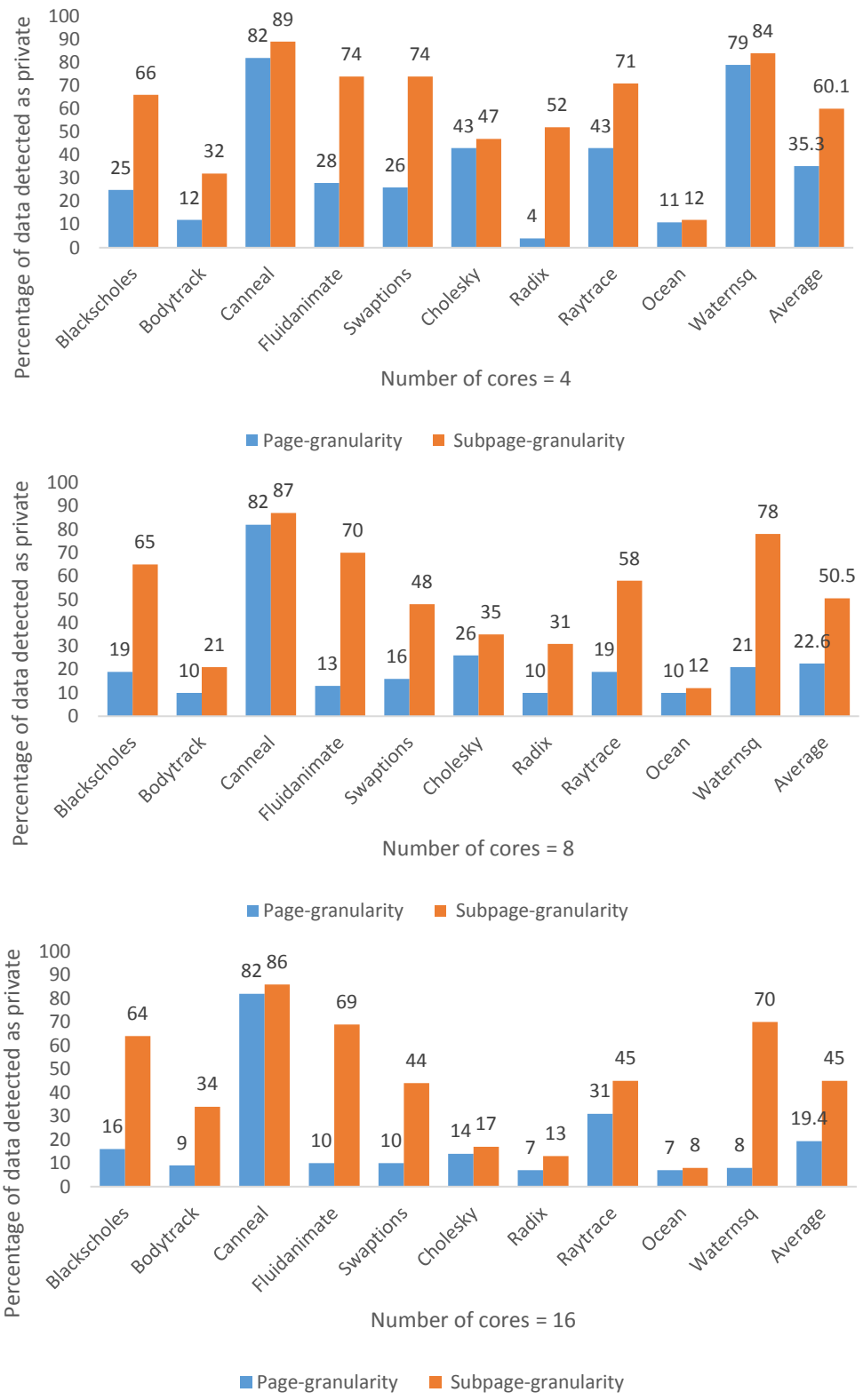

Figure 3.1: Percentage of data detected as private in a page granularity detection versus subpage granularity detection mechanisms.

in the second category need inquiring other TLBs for each missed page translation that happen in each core which can introduce both design complexity and also higher network traffic. For example, the work in [7] suggests snooping the other TLBs in the system for finding the page entry in other TLBs. This approach is not scalable with higher core counts due to traffic overhead and excessive energy consumption required by snooping.

This motivates us to propose a new method to boost virtual address translation that doesn't suffer from the shortcomings listed above. Our second observation motivates us to propose a technique to enhance the performance of virtual memory management.

Observation 2. *We can utilize the detected private data not only to mitigate the overhead of managing coherence in cache system, but also to improve the performance of the whole memory.*

Private data does not necessarily have all the requirements for shared data accesses. The most obvious example is the need to keep track of private data in directory caches. As will be shown, we can boost the performance of virtual to physical translation by introducing extra components that work in conjunction with private TLBs.

Chapter 4

Our Approach

In this chapter, we explain the details of our memory management scheme that employs a runtime subpage granularity private data detection motivated by our observations in Chapter 3. The two main mechanisms of our approach are i) the mechanism for detecting private memory blocks in subpage granularity and ii) the approaches that enable us to exploit the results of the data classification to improve the performance of the system.

4.1 Private Block Detection

A common approach to differentiate between private and shared data blocks is to utilize OS capabilities [50][47][49]. The prior work [50] extends TLB and page table entries with some additional fields to distinguish between private and shared pages. To do so, two new fields are introduced in TLB entries: while the private bit (P) indicates whether the page is private or shared, the locked bit (L) is employed to prevent race conditions when a private page becomes shared and in turn the coherence status of cache blocks belonging this particular page are restored. To distinguish between private and shared pages, three new fields are also attached to page table entries: the private bit (P) marks whether the

page is private or shared; if P is set, the keeper field indicates the identity of the unique core storing the page table entry in its TLB; cached-in-TLB bit (C) shows whether the keeper field is valid or not.

In our proposal, we extend the technique exploited in a previous by proposed work [50] to detect private data at *subpage granularity*. Like the prior work, we try to detect private data blocks at runtime, but unlike it, we intend to detect private data at subpage granularity. To accomplish this, we use most significant bits of page offset for subpage ID and clone V, P, C, L and keeper fields in TLB and page table entries so that each subpage has its own such fields, as depicted in Figure 4.1. In this work, we divide each page into a number of subpages. The size of the keeper field grows according to the number of cores in the system. In other words, the size of the keeper is $\log_2(N)$ where N is number of cores in the system. We should note that these extra storage requirements in page table entries are part of OS storage and does not force any additional storage requirement to the underlying hardware except the three bits required to identify the status of each subpage in TLB entry.

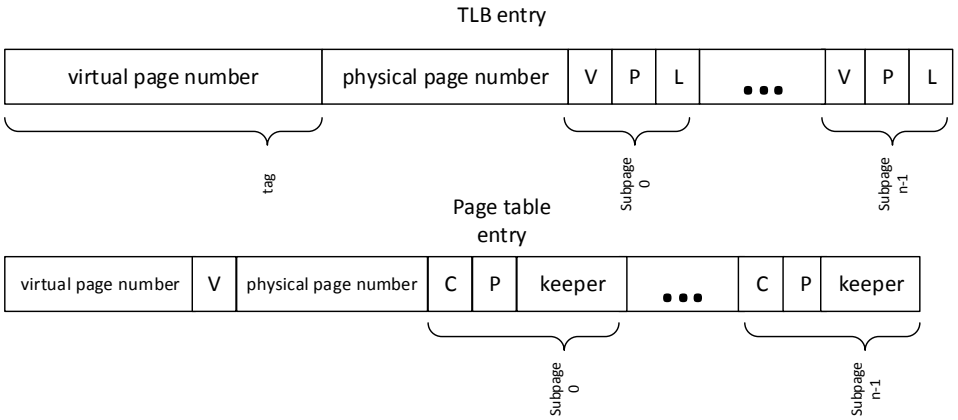

Figure 4.1: TLB and page table entry formats.

We, next, give the three main operations that should be performed to update the fields properly and enable detection of private data at subpage granularity. We also depict the operations in Figure 4.2.

- First (red): When a page is loaded into main memory for the first time,

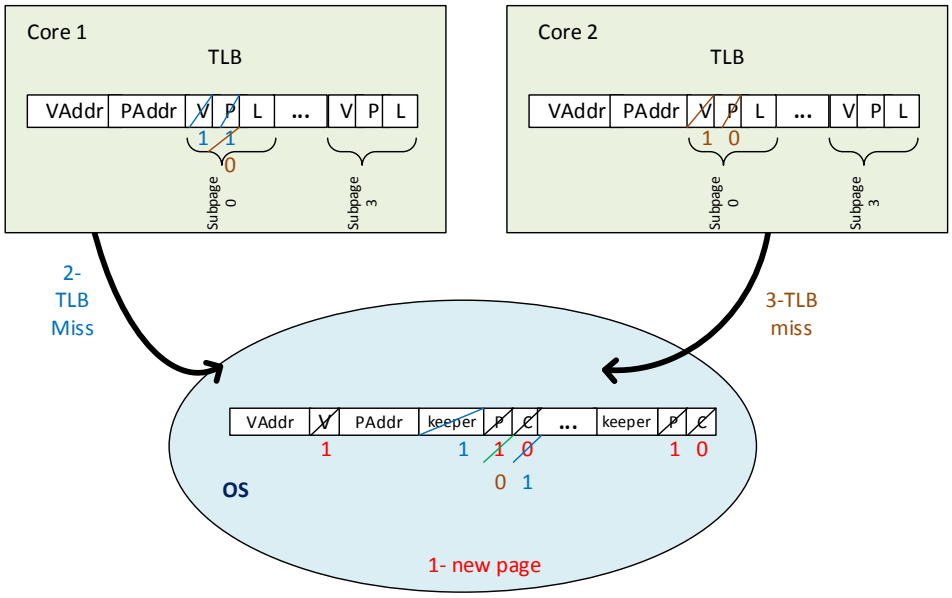

Figure 4.2: The subpage granularity private data detection mechanism.

the operating system allocates a new page table entry with the virtual to physical address translation. Besides storing the virtual to physical address translation in the page table entry, all subpages within that page are considered as private and thereby, the corresponding (P) bits are set. All subpages' bits (C) are also cleared, showing that the entry has not been cached in any TLB yet.

- Second (blue): Core1 faces a miss in its TLB for an address translation or there is a hit in the TLB but the (V) bit of the subpage which was tried to be accessed, is cleared (means it is not cached in TLB yet). In either cases, core1 will inquire the operating system page table for the translation of the subpage. It finds the (C) bit of the subpage cleared, which means that the subpage is not accessed by any other core yet. Thus, the (C) bit is set and the identity of the requester core (core 1) is recorded in the keeper field.

- Third (brown): Core2 experiences a miss in its TLB for the same subpage like the previous case. After looking up the page table for the specific subpage, it turns out both (C) and (P) bits of the subpage are set. Therefore, the keeper field should be compared against the identity of the requester

core. If there is a match, it means that the keeper core has already experienced a TLB miss and the page table entry is brought into the requester core's TLB, considering the subpage as privately accessed only by the requester core. If the keeper field does not match the identity of the core requesting the page table entry (like in this example), it means that two different cores are attempting to access the data within the same subpage (core 1 and core 2 in this example). In this situation, the operating system decides to turn the status of corresponding subpage to shared by clearing the (P) bit. Moreover, the operating system triggers the coherence recovery mechanism by informing the keeper core to restore the coherence status of cache blocks belonging to the subpage. We will explain the coherence recovery mechanism with more detail in the next section.

4.1.1 Coherence Recovery Mechanism

As will be seen in the coming sections, the performance improvement we expect to gain by our approach is based on the fact that we *avoid* applying some coherence operations for *private* data. In fact, some of the private data does not necessarily need many of the messages interchanged between the cache controllers. Similarly, the directory cache does not need to track private blocks like shared blocks.

Therefore, if at a certain point, we realize that our assumption about the private status of a block is no longer valid, then we need to recover from this situation. Otherwise, the caches might not remain coherent. In this work, we use a similar recovery mechanism proposed by a previous study [50]. In this work, authors propose two strategies, namely, flushing-based recovery and updating-based recovery mechanisms. Their results show that these two strategies are slightly different in terms of performance. Therefore, our recovery mechanism uses a similar flushing-based mechanism and performs following operations in order to ensure safe recovery from changing status of a subpage from private to shared. Figure 4.3 depicts the coherence recovery mechanism being applied in our proposal.

- First, the initiator core issues a recovery message (after trying to access a subpage in its TLB that is already being kept by another core in the system) to be sent to the keeper core. Before, completing the recovery, the initiator core should also lock that subpage in its TLB.

- Second, on the arrival of recovery request, the keeper core first should prevent accesses to the blocks of that subpage by setting the subpage's (L) bit.

- Third, the keeper should invalidate all the blocks corresponding to that subpage in its private cache. The keeper also should take care of the pending blocks in its Miss Status Holding Register (MSHR). If there are any blocks within that subpage in MSHR, they should be evicted right after the operation completes.

- Forth, once the mentioned operations finish, the keeper sends back an acknowledgment to announce the completion of the recovery. At this point, the core which initiated the recovery, change the subpage status of that specific subpage to shared and continue its operation.

4.2 Directory Cache Organization

In this section, we present our approach to address two most important drawbacks of directory-based cache coherence protocols. The first drawback is the need for a highly associative cache, which introduces high power consumption and high complexity in the design. Second, the high storage cost for keeping track of all the blocks that exist in the last level private caches. Unfortunately, both of these drawbacks threaten the scalability of the system. Moreover, the former linearly gets worse with the number of cores in the system as depicted in Chapter 2.

The primary solution for solving the discussed scalability problem inherited in the duplicate-tag directory based cache coherence protocols is adjusting the associativity value of directory cache to some low values similar to the ones in the

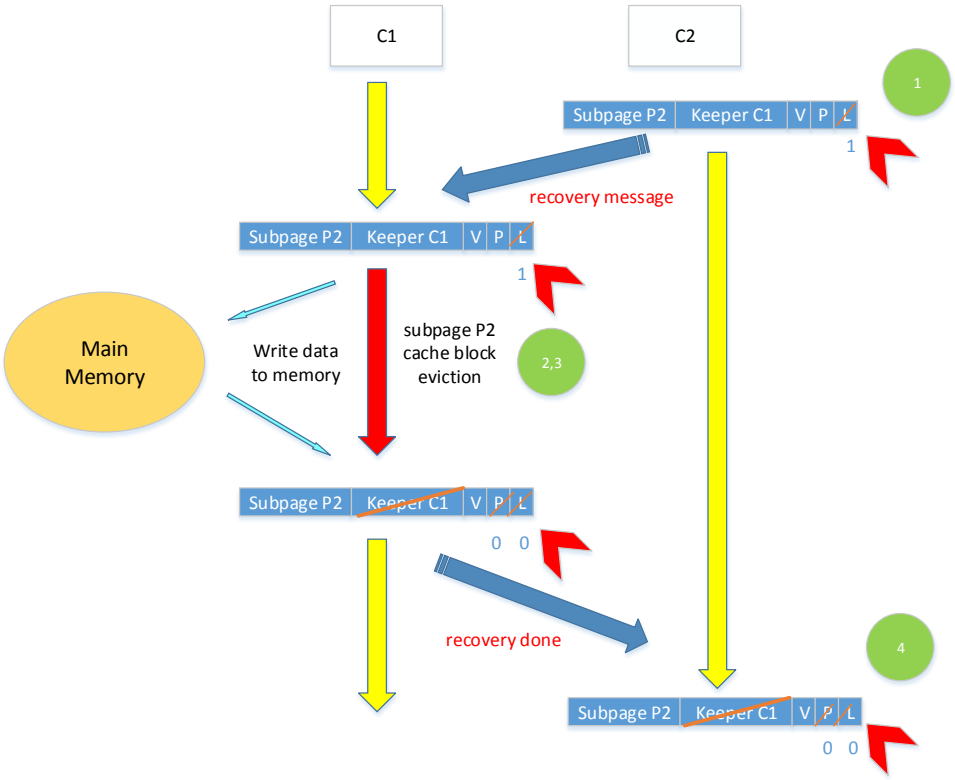

Figure 4.3: An example recovery mechanism shown for two cores, C1 and C2. Operations need to be proposed according to the order given in circles.

associativity of private caches. However, with this approach, the number of evictions in directory caches caused by adding a new entry to the directory cache might increase dramatically. Since any eviction in directory cache requires invalidation of all the copies of that block in the whole private caches in the system, this might jeopardize the performance of the system. Thus, the directory cache has the potential to become a limiting factor in performance of large-scale many-core architectures.

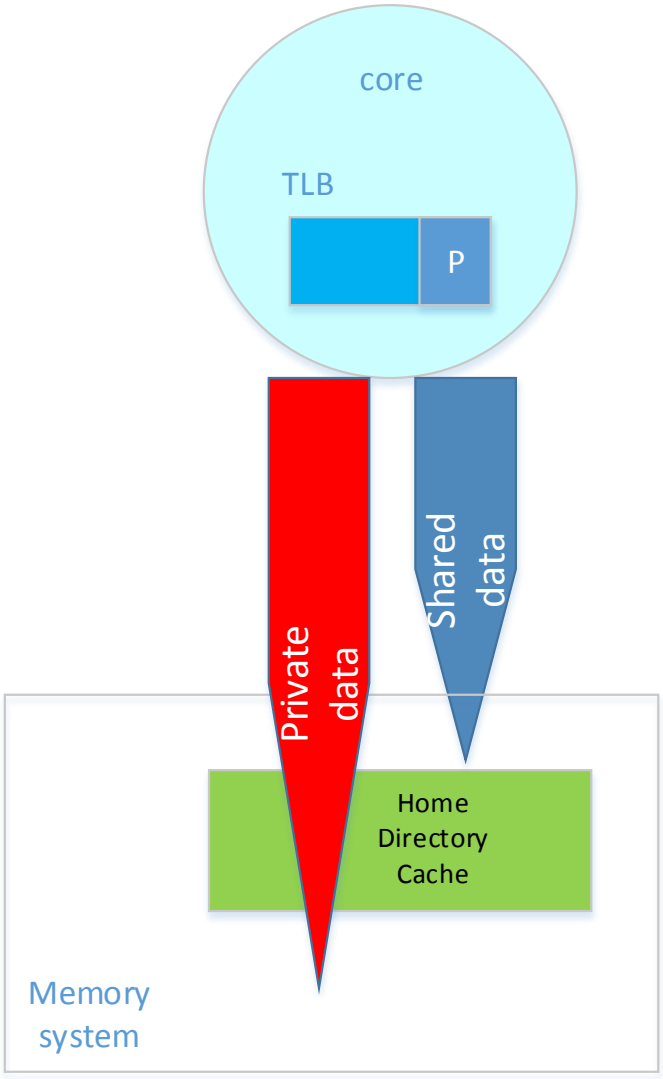

Figure 4.4: Directory cache organization in our proposal.

In this work, we try to utilize our private data detection to address the aforementioned scalability problem as follows. We avoid polluting the directory cache with private data by holding the states for the shared data but not for the private ones. Figure 4.4 shows directory organization applied in our proposal and how private data must avoid accessing the home directory cache while shared data need a directory access before issuing further coherence requests. The decision about private and shared status of a block is made based on the status of the subpage (P bit) that a block belongs to. As we will show in the evaluation section, we can dramatically decrease the directory cache eviction rate and mitigate the inevitable performance degradation due to high directory eviction counts. The invalidation of the blocks related to the evicted directory entries is performed as normal. In the sensitivity analysis, we show that our idea works with different associativity values and we get acceptable performance results even for directory caches with low associativity.

4.3 On-chip Page Table

Memory classification at subpage level may increase the frequency of the operating system's involvement in updating the maintenance bits belonging to subpages stored in page table entries, nullifying some portion of performance benefits of subpage level data classification. For this reason, as our second contribution, we show how we can negate the possible performance degradation by introducing on-chip page table. Moreover, the proposed method also enables us to boost the performance of the virtual memory management in many-core systems. Based on the drawbacks of previous studies on TLB organization (see Chapter 2), we propose our on-chip page table as follows.

1- We increase the probability of accessing a page translation within the chip by introducing excessive capacity rather than the private TLBs for keeping the page translation in the chip. In our implementation, each core's private TLB includes 64 entries, thereby using the same size on-chip page table per core. We reserve this space for our *on-chip page table* by not keeping the private data status in

directory cache. In the experimental results, we show how devoting even a small portion of the directory cache to the page translation can make the translation faster without adversely affecting the performance of other components.

2- In contrast to the discussed approaches, we avoid snooping by distributing the pages based on their tags into the on-chip page tables located in the directory controller. Distribution is done by interleaving the page entries according to the least significant bits of virtual page numbers (for example, with 16 cores, we use 4 least significant bits). This way, for each miss in the private TLBs, the core sends the request for the page address translation only to one of the cores' on-chip page table. This makes our methods more scalable compared to other proposed cooperative TLBs based on snooping.

Figure 4.5 depicts the structure of our on-chip page table and how a miss in one of the private TLBs can be resolved by one of the on-chip page tables. After a TLB miss occurs in core 0 for an address translation of page 'a', the request for finding the page information for page 'a' is sent directly to the core which may have the entries for that page (in this example core N-1). Then, the translation is forwarded back to the requested core in case it is found in the on-chip page table (red line). However, in the second case, the search for finding the translation for page 'b' was not successful. Therefore, with OS involvement, the entries will be written to one of the on-chip page tables after interleaving (core 1) and also to the TLB of the requested core (core 0).

In our experiments, we show how much can be avoided referring to OS thanks to exploiting the on-chip page table. We also show the effectiveness of our approach with varying TLB sizes.

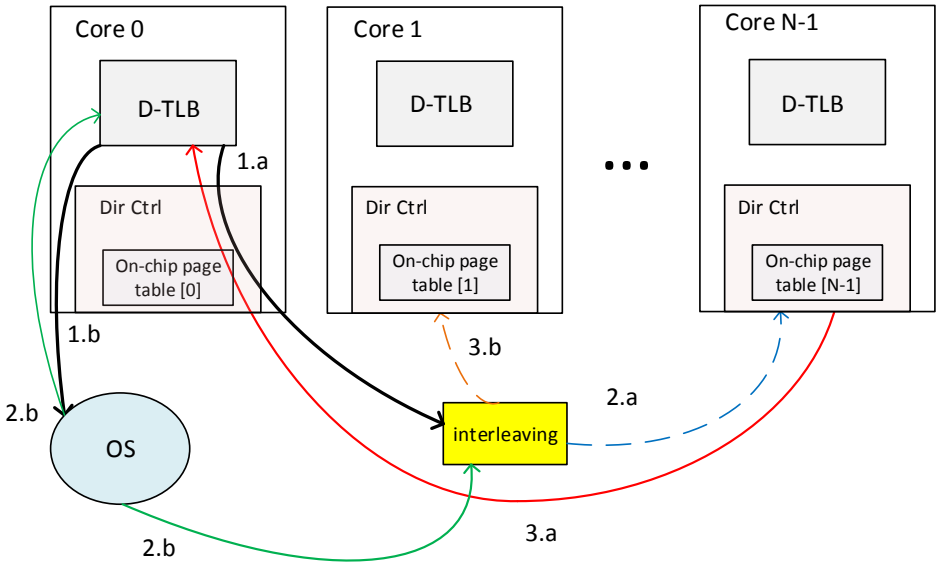

Figure 4.5: The structure of on-chip page table in a CMP with private per-core TLBs.

Chapter 5

Methodology

5.1 System Setup

We evaluate our proposal with Gem5 full-system simulator [53] running Linux version 2.6. Gem5 uses RUBY which implements a detailed model for the memory subsystem and specifically cache coherence protocol. For modeling the interconnection network, we use GARNET [54], a detailed interconnection simulator also included in gem5. We apply our idea to *MOESI-CMP-Directory* which is a directory-based cache coherence protocol implemented in gem5. We present results with a system consisting of 16 cores with level one private data and instruction caches, and a shared level two cache. Table 5.1 provides more details on our simulation environment. In the rest of this thesis, this configuration is considered as the base setup, where our approach based on private/shared data classification is not applied.

Table 5.1: Default simulation parameters used in our experiments.

Processor	16 Alpha cores, 2GHz, 64 entries TLB
Private L1D	32 KB (=**512** entries), 4-way associative, 64B cache-block size
Private L1I	32 KB, 2-way associative
Shared L2	32MB (2MB per core) , 8-way associative
Directory cache	**512** entries, ***4-16*** way associative
Network	Torus , Fixed Garnet interconnection model
Cache coherence protocol	MOESI CMP Directory
Page size	8KB

5.2 Benchmarks

We evaluate our proposal with ten different parallel workloads from two commonly used suites (SPLASH-2 [55] and PARSEC 2.1 [56]). SPLASH and PARSEC are two commonly used benchmark suits in research. They have been parallelized to take advantage of multiprocessors. The applications within these two suits include a wide range of application domains as indicated in Table 5.2. As indicated in the Table 5.3, we simulated the applications mostly with small size data sets since we executed a large set of simulations to provide a comprehensive sensitivity analysis, and each simulation takes a considerable amount of time to run in our full-system simulator.

For our experimental results, we only consider the parallel phase of benchmarks or Region of Interest (ROI); and the number of threads used by each application is set based on the number of cores in the system, which is 16 by default.

5.3 Evaluation Metrics

We compare our approach with a state-of-the-art directory-based cache coherence protocol. The specific metrics we tested are the number of evictions in the directory cache, cache miss ratio, cache miss latency, the network traffic, and the

Table 5.2: Benchmarks and respective input files.

Benchmarks	Application Domain
Parsec 2.1	
Blacksholes	Financial Analysis
Bodytrack	Computer vision
Canneal	Engineering
Fluidaniate	Animation
Swaptions	Financial Analysis
Splash 2	
Cholesky	High-Performance Computing (matrix factorization)
Raytrace	Graphics
Waternsq	High-Performance Computing (molecular dynamics)
Radix	General (sort technique)
Ocean	Scientific Computation (Oceanography)

execution time.

These metrics help us evaluate the scalability of our approach against the base setup. We observe how much of the evictions can be avoided in directory cache and the respective effects on other metrics. For instance, fewer evictions in directory cache results with less number of cache block invalidations. Therefore, miss ratio is expected to change in our proposal.

Another metric, we used to demonstrate the performance of our proposal is cache miss latency. Our proposal not only affects the cache miss ratio, but it is also able to reduce the average cache miss latency, due to the fact that a directory cache lookup for private blocks is avoided in our proposal. In the next chapter, we explain how much our proposal can boost the performance in terms of cache miss latency.

Moreover, we examine the network coherence traffic as another important metric in evaluation of cache coherence management. The reduction in cache misses and directory eviction directly affect the network traffic as will be shown in the next chapter.

Finally, we evaluate the overall performance of our proposal against the base

Table 5.3: Benchmarks and respective input files.

Benchmarks	Input
Parsec 2.1	
Blacksholes Bodytrack Canneal Fluidaniate Swaptions	simsmall
Splash 2	
Cholesky	tk15.O
Raytrace	Teapot environment
Waternsq	512 molecules, 3 time steps
Radix	1048576 keys
Ocean	258*258 ocean

setup, but by comparing the execution latencies for all set of the benchmarks.

As explained earlier, for a directory cache based on duplicate-tag, the associativity of directory cache and its size are the limiting factors for scalability. Since scalability is a major concern in this thesis, we test our approach against the state-of-the-art directory cache coherence protocol by performing a sensitivity analysis.

Chapter 6

Evaluation

In this chapter we show how our proposal is able to improve the performance of directory-based cache coherence protocol through reducing cache miss ratio, network traffic, and latency of resolving cache misses. Therefore, we first show the reduction in terms of evictions in the directory cache. Then, we show the effects in cache miss ratio as well as the number of messages interchanged in the network as a result of better directory cache eviction rate. We also study the performance of our proposal in reducing the latency of resolving a cache miss. Moreover, we evaluate our on-chip page table by showing the amount of misses in private TLBs that can be resolved by introducing our on-chip page table. Finally, we discuss the effects of using directory caches with low associativity.

6.1 Directory Cache Eviction

As discussed earlier in this thesis, a considerable amount of accessed memory blocks are private; and therefore, we do not keep track of those blocks in directory caches. By not polluting the directory cache with status of private blocks that do not need coherence maintenance, we aim to have less evictions in directory cache even for caches with lower associativity. Figure 6.1 shows directory cache

eviction rate for our proposal normalized with respect to the base configuration. As can be seen from this Figure, on average, we have 58% less evictions in the directory cache compared to the base setup without any data classification.

The reduction in directory cache eviction ratio can be the result of two factors. First, the percentage of detected private data blocks, and second, the access pattern in which shared blocks are accessed. Arguably, if we can detect more private data, we can avoid more evictions in the directory caches. For instance, number of directory evictions has been reduced for Waternsq more than Cholesky, since, we were able to detect higher number of private data blocks for Waternsq compared to Cholesky (70% for Waternsq and 17% for Cholesky). However, it is not the only factor which affects reduction in the directory eviction. The sharing pattern of an application can also play an important role in the number of evictions that happen in the directory. For instance, for Canneal, we were able to detect a high percentage of private data blocks (86%) compared to Bodytrack (34%). But, because of the difference in sharing pattern of these two applications, we observe rather same normalized eviction rates. The reason is due to the temporal communication behavior of these two applications. Previous research [1] observed that, in Canneal, the communication between the cores take place throughout the execution of the application, whereas, in Bodytrack, for the majority of the parallel phase, there is limited communication between cores as shown in Figure 6.2. Therefore, in Canneal, the chances for a possible conflict will be higher than Bodytrack.

Another observation in Figure 6.1 is the dramatic improvement in directory eviction ratio for Waternsq application. Based on the results reported in [1], Waternsq and Waterspa that involve *all cores* actively in a producer-consumer pattern. Furthermore, based on these observation, they conclude that a broadcast-based technique is likely to benefit from this in Waternsq and Waterspa. Therefore, we also observe that Waternsq benefits the most.

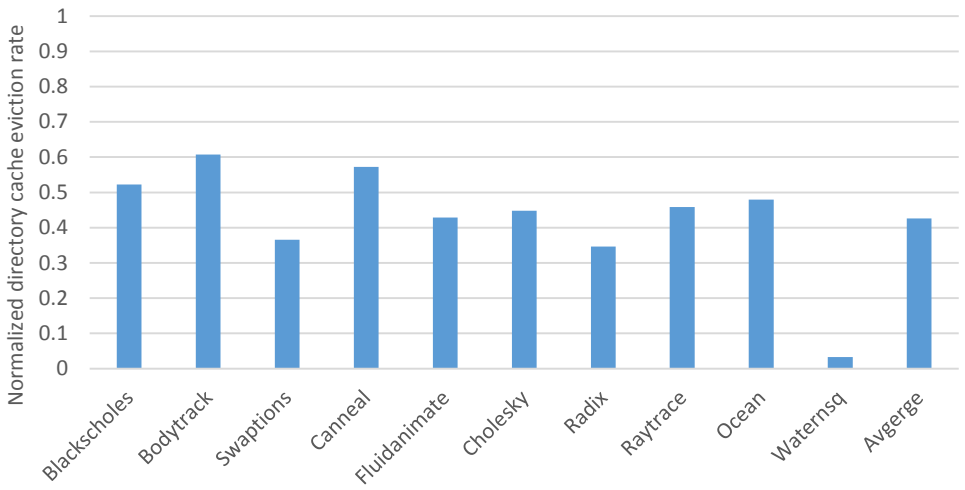

Figure 6.1: Normalized directory cache eviction rate.

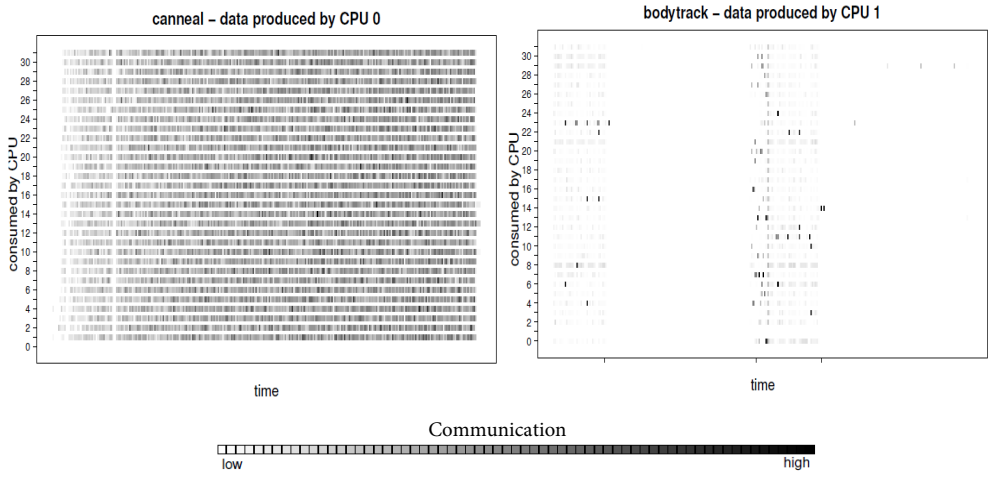

Figure 6.2: Normalized communication changes over time for Bodytrack and Canneal applications [1].

6.2 Private Cache Misses

One of the primary advantages of reducing directory cache eviction is the reduced invalidations at the last level private caches (in our system, the L1 cache). This is due to the fact that any eviction of a block in directory cache implies invalidation of all the blocks in any of the L1 caches that correspond to that block. Figure 6.3 shows the L1 cache miss ratio for 10 different multi-threaded applications normalized with respect to the base case. As can be seen from this figure, our approach reduces the private L1 cache miss ratio by 15%, on average.

Moreover, we generate better results for those applications with fewer misclassified blocks. In other words, the higher the number of private blocks detected out of all actual private data blocks exist, the higher the reduction is in the number of cache misses.

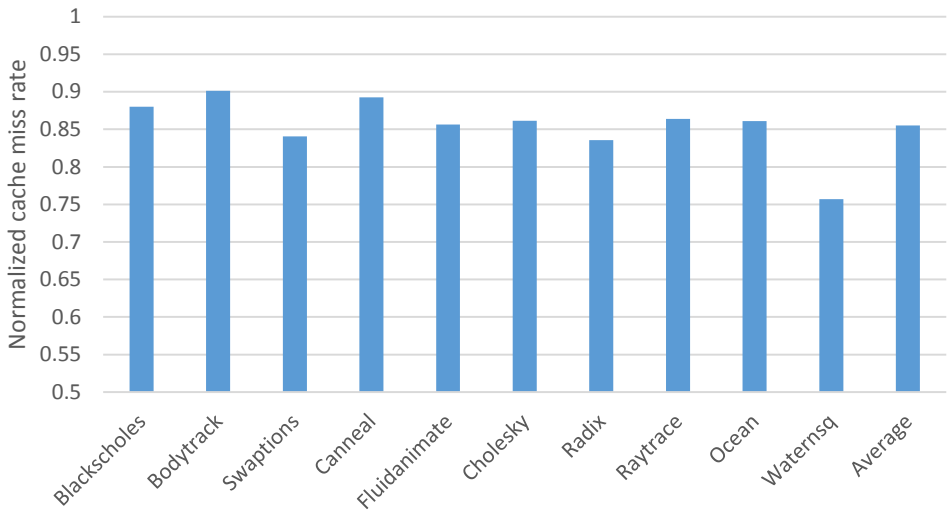

Figure 6.3: Normalized L1 cache miss rate.

6.3 Network Traffic

The number of messages exchanged in the system is also reduced by our proposal. This is because evictions in directory caches and processor cache misses impact the

network message counts. For instance, for resolving a miss in the first level private cache, different controllers in the system need to exchange request, forward, and response messages with one another. In Figure 6.4, we compare the message counts of our proposal normalized with respect to the base system. Our approach reduces the message traffic between 9% and 21% with an average reduction of 12%.

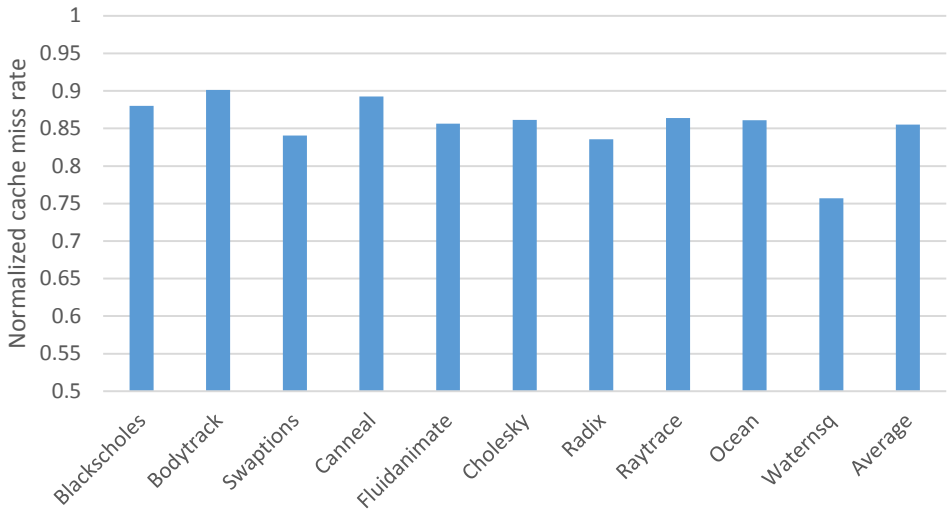

Figure 6.4: Normalized network traffic message count.

6.4 Cache Miss Latency

With our proposal, we are also able to reduce the average latency for resolving cache misses. Normally, in a CMP, when a core experiences a miss in its cache, it tries to find the data block with right permission in other cores' caches. For inquiring those permission information, a directory lookup is required. However, for the case of private block, particularly, it is known that no other core cached that block of data. So, we may avoid referring to directory cache for those requests associated to private blocks. Therefore, some portion of the requests experience less latency with a cache miss, lowering the overall average latency of a cache miss. Figure 6.5 depicts the average cache miss latency normalized to the base

case for our applications. As can be seen, on average, we resolve cache misses 8% faster than the base system.

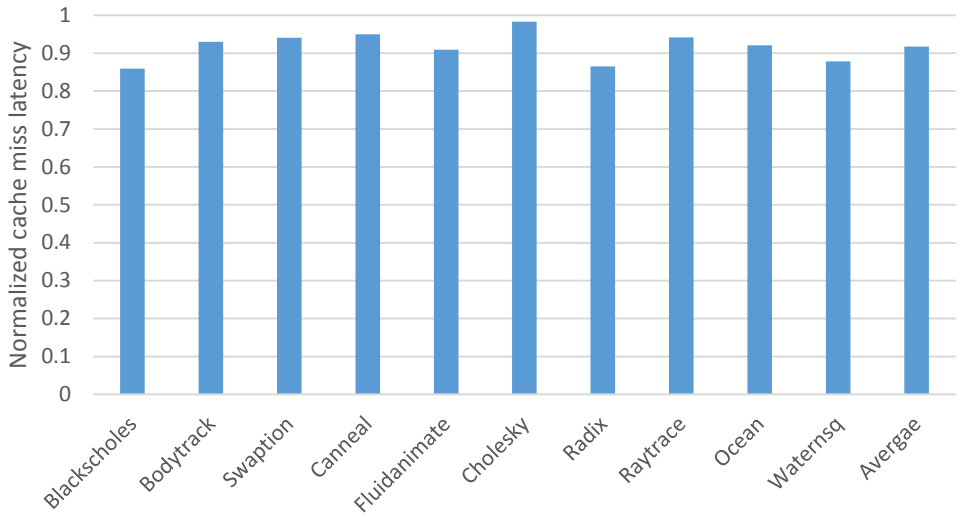

Figure 6.5: Normalized cache miss latency.

6.5 Coherence Recovery Overhead

To demonstrate that the coherence recovery has negligible effect on the performance of our proposal, Figure 6.6 shows the times that recovery mechanism has been triggered in proportion to total TLB accesses, throughout the execution of each application. As it can be seen, on average, 1.34% of accesses to TLB caused a coherence recovery to be triggered.

6.6 Execution Time and Sensitivity Analysis

The reduction in the number of cache misses, the reduced latency in resolving them, and a decline in the number of messages exchanged in the network; all result in a reduction in the latency of executing the application as it is shown in Figure 6.7. This figure also illustrates the sensitivity of our approach to the

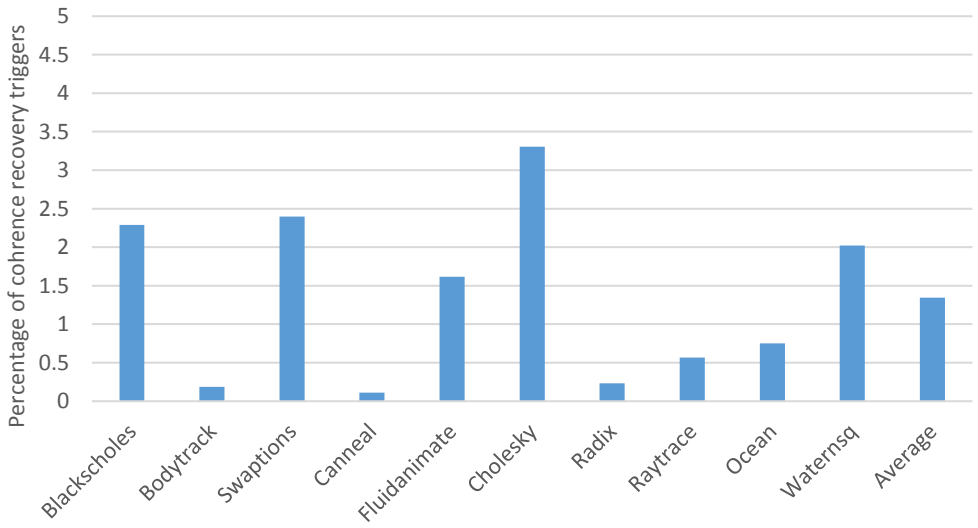

Figure 6.6: Percentage of TLB accesses that triggered the coherence recovery.

associativity of directory cache. The three bars labeled as DS-Assoc-4, DS-Assoc-8, DS-Assoc-16 represent three configurations with directory cache associativity equal to 4, 8, and 16, respectively. On average, we generate 4%, 6%, and 7% improvement in execution time for directory caches with associativity equal to 4, 8, and 16, respectively. As it was discussed in early chapters, high-associativity requirement for directory caches is a major problem for scalability, especially with high core counts. The key observation in Figure 10 is that our approach enables quite similar results even for directory caches with lower associativity.

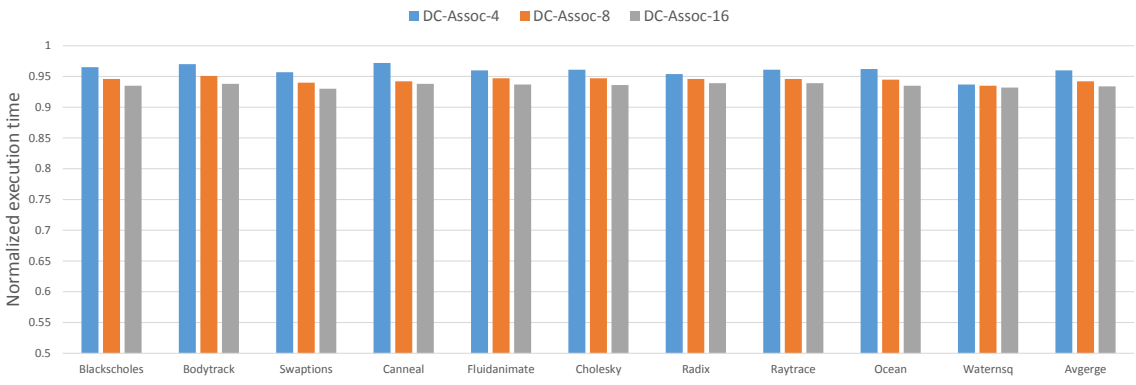

Figure 6.7: Execution time normalized with respect to the base system. DC-Assoc-4, DC-Assoc-8, and DC-Assoc-16 stand for directory caches with associativity equal to 4, 8, and 16, respectively.

6.7 Performance of On-Chip Page Table

We, next, show the improvements of virtual memory management by introducing the on-chip page table. Figure 6.8 shows the percentage of TLB misses that also experience a miss in on-chip page tables. In other words, it shows how much we can avoid accessing the costly OS page table by finding the required page translation in the distributed page table. As an example, for Canneal benchmark, only 8% of page translations which causes a miss in private TLBs, can not be found in the on-chip page table. Therefore, the remaining 92% of accesses find the right translation in the on-chip page table after they experienced a miss in private TLBs. On average, our approach prevents 84% of accesses to operating system by introducing the on-chip page table.

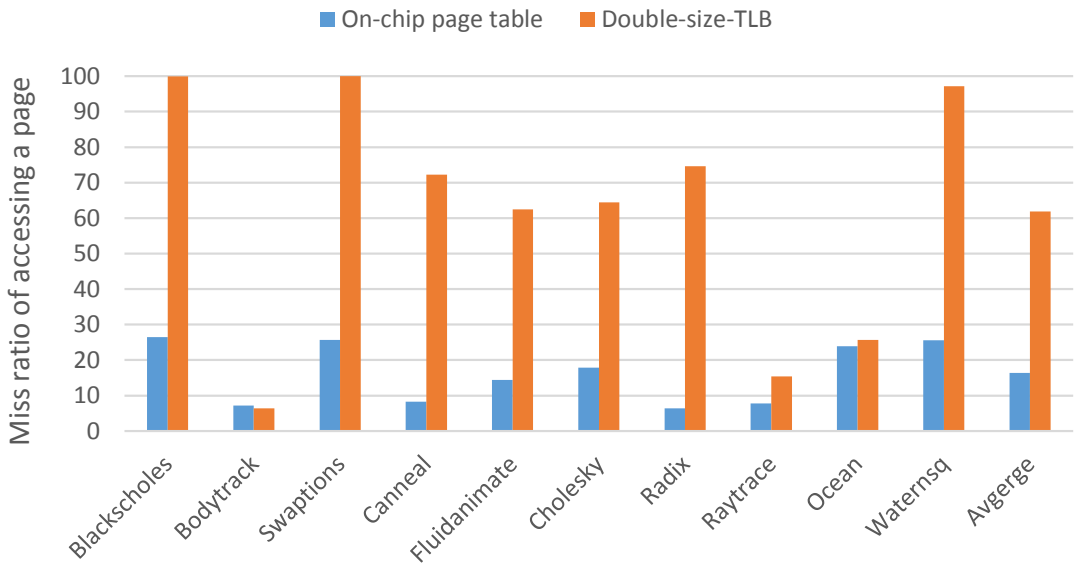

Figure 6.8: Normalized miss rate of finding page translation in a core.

We also compared our on-chip page table with a modified version of the base system, where the TLB is double the original size. As can be seen in Figure 6.8, by doubling the TLB size, we can only eliminate the 39% of the accesses to OS page table. So, we can conclude that increasing the cache capacity for page translation can not solely improve the performance. Moreover, exploiting an effective technique which enables sharing page translation between the cores

is also very crucial for providing a fast virtual page translation. As it can be seen, in most of the benchmarks, there is a big difference between the miss rate for on-chip page table and double-size TLB which proves the former statement.

Chapter 7

Conclusion and Future Work

In this thesis, we propose a subpage granularity data management scheme which improves the performance of directory-based cache coherence protocols by decreasing the number of evictions taking place in directory caches. This is done by applying a subpage granularity data classification which helps us not to keep track of significant percentage of data blocks in directory caches. We also accelerate virtual to physical address translation by introducing on-chip page table. Specifically, we avoid 84% of accesses to the OS page table. Overall, we observe up to 7% improvements in execution time even for directory caches with lower associativities, which ensures the scalability of our approach.

We may extend the proposal in this thesis by applying new private data detection mechanisms to detect more of the private data. One way of doing so is by applying a compiler optimization pass which detects some of the private data statically at compile time. The compiler-assisted detection mechanism can cooperate with our runtime subpage granularity detection approach to enable detection of more private data without introducing any additional overheads to the hardware.

Bibliography

[1] N. Barrow-Williams, C. Fensch, and S. Moore, "A Communication Characterisation of Splash-2 and Parsec," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 86–97, IEEE Computer Society, 2009.

[2] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan 1998.

[3] J. B. Chen, A. Borg, and N. P. Jouppi, "A simulation based study of tlb performance," *SIGARCH Comput. Archit. News*, vol. 20, pp. 114–123, Apr. 1992.

[4] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, (Washington, DC, USA), pp. 195–206, IEEE Computer Society, 2002.

[5] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based tlb preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, (New York, NY, USA), pp. 117–127, ACM, 2000.

[6] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," *SIGPLAN Not.*, vol. 29, pp. 171–182, Nov. 1994.

[7] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *Microarchitecture (MICRO),*

2010 43rd Annual IEEE/ACM International Symposium on, pp. 313–324, Dec 2010.

[8] D. Lustig, A. Bhattacharjee, and M. Martonosi, "Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 2:1–2:38, Apr. 2013.

[9] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pp. 29–40, Sept 2009.

[10] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 62–63, Feb 2011.

[11] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, pp. 66–76, Dec. 1996.

[12] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, pp. 690–691, Sept. 1979.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 18, pp. 15–26, May 1990.

[14] L. A. Barroso and M. Dubois, "The performance of cache-coherent ring-based multiprocessors," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), pp. 268–277, ACM, 1993.

[15] A. Ahmed, P. Conway, "AMD opteron shared memory mp systems," In Proceedings of the 14th HotChips Symposium, Aug. 2002.

[16] D. Kanter, "The common system interface: Intels future interconnect," Intels Future Interconnect. http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032.

[17] M. M. K. Martin, "Token coherence," PhD thesis, University of Wisconsin, 2003.

[18] L. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *Computers, IEEE Transactions on*, vol. C-27, pp. 1112–1118, Dec 1978.

[19] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, (New York, NY, USA), pp. 749–753, ACM, 1976.

[20] S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb, "The alpha 21364 network architecture," in *Hot Interconnects 9, 2001.*, pp. 113–117, 2001.

[21] J. Laudon and D. Lenoski, "The sgi origin: A ccnuma highly scalable server," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (New York, NY, USA), pp. 241–251, ACM, 1997.

[22] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato, "A two-level directory architecture for highly scalable cc-numa multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, pp. 67–79, Jan. 2005.

[23] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium on*, pp. 280–289, May 1988.

[24] D. Abts, S. Scott, and D. Lilja, "So many states, so little time: verifying memory coherence in the cray x1," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pp. 10 pp.–, April 2003.

[25] S. S. Mukherjee and M. D. Hill, "An evaluation of directory protocols for medium-scale shared-memory multiprocessors," in *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, (New York, NY, USA), pp. 64–74, ACM, 1994.

[26] B. O'Krafka and A. Newton, "An empirical evaluation of two memory-efficient directory methods," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 138–147, May 1990.

[27] R. Simoni, "Cache coherence directories for scalable multiprocessors," Ph.D. Thesis, Stanford University, 1992.

[28] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st ed., 2011.

[29] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," *SIGARCH Comput. Archit. News*, vol. 35, pp. 46–56, June 2007.

[30] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," *SIGPLAN Not.*, vol. 26, pp. 224–234, Apr. 1991.

[31] R. Simoni, M. A. Horowitz, "Dynamic pointer allocation for scalable cache coherence directories," in: Intl Symp. on Shared Memory Multiprocessing, April 2001, pp. 7281.

[32] R. Simoni, "Cache coherence directories for scalable multiprocessors," Ph.D. Thesis, Stanford University, 1992.

[33] B. O'Krafka and A. Newton, "An empirical evaluation of two memory-efficient directory methods," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 138–147, May 1990.

[34] A. Ros, M. Acacio, and J. Garc?a, "A novel lightweight directory architecture for scalable shared-memory multiprocessors," in *Euro-Par 2005 Parallel Processing* (J. Cunha and P. Medeiros, eds.), vol. 3648 of *Lecture Notes in Computer Science*, pp. 582–591, Springer Berlin Heidelberg, 2005.

[35] A. K. Nanda, A. Nguyen, M. M. Michael, D. J. Joseph, "High-throughput coherence control and hardware messaging in everest," IBM Journal of Research and Development 45 (2) (2001) 229244.

[36] J. Kong, P. Yew, G. Lee, "Minimizing the directory size for large-scale shared-memory multiprocessors," IEICE Transactions on Information and Systems E88-D (11) (2005) 25332543.

[37] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A scalable architecture based on single-chip multiprocessing," *SIGARCH Comput. Archit. News*, vol. 28, pp. 282–293, May 2000.

[38] H. Hossain, S. Dwarkadas, and M. Huang, "Pops: Coherence protocol optimization for both private and shared data," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 45–55, Oct 2011.

[39] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "Spatl: Honey, i shrunk the coherence directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 33–44, Oct 2011.

[40] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 423–434, ACM, 2009.

[41] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving multiprocessor performance with coarse-grain coherence tracking," *SIGARCH Comput. Archit. News*, vol. 33, pp. 246–257, May 2005.

[42] J. Zebchuk, E. Safi, and A. Moshovos, "A framework for coarse-grain optimizations in the on-chip memory hierarchy," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 314–327, IEEE Computer Society, 2007.

[43] Y. Li, R. Melhem, and A. Jones, "A practical data classification framework for scalable and high performance chip-multiprocessors," *Computers, IEEE Transactions on*, vol. 63, pp. 2905–2918, Dec 2014.

[44] L. Jin and S. Cho, "Sos: A software-oriented distributed shared cache management approach for chip multiprocessors," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, (Washington, DC, USA), pp. 361–371, IEEE Computer Society, 2009.

[45] S. Shao, A. Jones, and R. Melhem, "Compiler techniques for efficient communications in circuit switched networks for multiprocessor systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, pp. 331–345, March 2009.

[46] Y. Li, R. Melhem, and A. Jones, "Leveraging sharing in second level translation-lookaside buffers for chip multiprocessors," *IEEE Comput. Archit. Lett.*, vol. 11, pp. 49–52, July 2012.

[47] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: Near-optimal block placement and replication in distributed caches," *SIGARCH Comput. Archit. News*, vol. 37, pp. 184–195, June 2009.

[48] C. Kim, D. Burger, and S. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *Micro, IEEE*, vol. 23, pp. 99–107, Nov 2003.

[49] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 111–122, ACM, 2010.

[50] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 93–104, ACM, 2011.

[51] H. Zeffer, Z. Radović, M. Karlsson, and E. Hagersten, "Tma: A trap-based memory architecture," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, (New York, NY, USA), pp. 259–268, ACM, 2006.

[52] C. Fensch and M. Cintra, "An os-based alternative to full hardware coherence on tiled cmps," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 355–366, Feb 2008.

[53] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[54] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 33–42, April 2009.

[55] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, (New York, NY, USA), pp. 24–36, ACM, 1995.

[56] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.