Optimized Coherence Tracking and Private Caching in Many-core Server Processors

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Sudhanshu Shukla

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

April, 2019

CERTIFICATE

It is certified that the work contained in the thesis entitled "***Optimized Coherence Tracking and Private Caching in Many-core Server Processors***", by ***Sudhanshu Shukla***, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Mainak Chaudhuri

Dr. Mainak Chaudhuri

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

November, 2018

Synopsis

The cache-coherent many-core server processors are traditionally designed with a per-core private cache hierarchy and a large shared multi-banked last-level cache (LLC). The private cache contents are kept coherent with the help of a scalable cache coherence protocol. An important storage structure employed by the cache coherence protocol is the sparse directory, which keeps track of the location(s) of the cache blocks in the private caches. The height or the number of entries in the sparse directory puts an upper bound on the number of cache blocks that can be tracked simultaneously at any point in time. The width of the sparse directory puts an upper bound on the number of simultaneous sharers of a cache block depending on the encoding format of the sharers. Even after decades of research, design of a space-efficient sparse directory that can scale seamlessly and offer high performance without curtailing the degree of sharing remains one of the most important problems in cache-coherent many-core server processors. In this thesis, we present novel solutions for optimizing the width and height of the sparse directory. In addition to the space overhead of coherence tracking, a many-core processor suffers from large round-trip latency overhead and traffic overhead in the interconnect between the per-core private cache hierarchy and the shared LLC banks. The cache coherence protocol, the sparse directory organization, and the efficiency of the private cache hierarchy together determine the latency and traffic overhead. While our sparse directory proposals exhibit significant savings in the interconnect traffic, we propose a more direct solution to this important problem by taking a fresh look at the private cache hierarchy architecture in the context of popular server workloads. In this thesis, we propose a novel space-efficient private cache hierarchy architecture that outperforms the traditional private cache hierarchy designs in

terms of interconnect traffic and execution time.

Our width-optimized sparse directory proposal exploits the observation that multi-threaded applications typically require two types of directory entries, namely, limited pointer entries tracking a few sharers of a block and bitvector entries tracking larger number of sharers for widely shared blocks. While this is a well-known behavior, the difficulty in employing this observation to practice in a space-efficient manner arises from the fact that the exact proportion of these two types of entries varies across applications and across phases even within the same application. Recent proposals aiming to optimize the average number of bits per directory entry have organized the sparse directory as either a static mix of these two types of entries with a pre-defined proportion or a collection of relatively short bitvector entries that can encode either a limited number of sharer pointers or a larger number of sharers hierarchically. In either case, sparse directory space is wasted depending on the run-time characteristics of the application. We present a directory organization that facilitates allocation of the two different types of directory entries dynamically. Our design maintains a pool of limited pointer entries, where each entry can also double as a segment directory entry encoding the sharers in a cluster of cores. Each tag in the primary sparse directory array has a pointer that can either represent a sharer or point to an entry in the pool. When multiple segment directory entries are needed to encode all the sharers of a block, our pool management protocol guarantees that all these entries are allocated contiguously so that maintaining a pointer to the head entry is enough. Detailed simulation results show that our Pool Directory proposal incorporated in a 128-core system running multi-threaded applications drawn from scientific, general-purpose, and commercial computing domains can offer, on average, 5% improvement in performance and 20% savings in interconnect traffic compared to the state-of-the-art scalable coherence directory (SCD) proposal when using a $\frac{1}{16}\times$ sparse directory.

Recent notable research efforts for optimizing the sparse directory height have largely followed two directions. First, the observation that private coarse-grain regions can be tracked in a single directory entry has motivated researchers to explore support for multi-

grain coherence. Second, pages or blocks have been identified dynamically as private respectively by the operating system (OS) or the hardware. The blocks thus marked private are not tracked by the directory saving directory entries. However, when such a block is shared at a later point, a costly non-scalable broadcast-based recovery mechanism is needed. In this thesis, we design a robust minimally-sized sparse directory that can offer adequate performance while enjoying the simplicity, scalability, and OS-independence of traditional broadcast-free block-grain coherence. We begin our exploration with a naïve design that does not have a sparse directory and the location/sharers of a block are tracked by borrowing a portion of the block's LLC data way. Such a design, however, lengthens the critical path from two transactions to three transactions (two hops to three hops) for the blocks that experience frequent shared read accesses. This is because the LLC data way cannot provide the block in such cases as part of the LLC block is corrupted with tracking information. We address this problem by architecting a tiny sparse directory that dynamically identifies and tracks a selected subset of the blocks that experience a large volume of shared accesses. However, it is difficult to appropriately size this tiny directory, since the actual footprint of the shared blocks varies across applications and across phases even within the same application. Therefore, to make the design robust, we further augment the tiny directory proposal with an option of selectively spilling directory entries into the LLC space for tracking the coherence of the critical shared blocks that the tiny directory fails to accommodate. Our Tiny Directory proposal operating with $\frac{1}{32}\times$ to $\frac{1}{256}\times$ sparse directories offers performance within a percentage of a traditional $2\times$ sparse directory in a 128-core system. The Tiny Directory design outperforms the state-of-the-art by very large margins when operating with the same number of directory entries.

Traditionally, the private cache hierarchy in the many-core server processors treats the private and the shared blocks equally. Our private cache hierarchy design proposal is motivated by the observation that in a single-level private cache hierarchy with per-core private L1 cache, elimination of all non-compulsory non-coherence L1 cache misses to a small subset of read-shared code and data blocks can save a large fraction of the LLC accesses indicating large potential for reducing the interconnect traffic in such architectures. These read-shared code and data blocks should be protected from eviction for a longer period of time in the private cache hierarchy. We architect a specialized exclusive per-core private L2 cache which serves as a victim cache for the per-core private L1 cache. The proposed victim cache selectively accommodates a subset of the L1 cache victims and manages the victim cache contents with specialized replacement policies so that the target subset of the read-shared blocks enjoys longer residence in the private cache hierarchy. The selective victim caching and the replacement policy proposals are driven by an online partitioning of the L1 cache victims based on two distinct features, namely, an estimate of sharing degree and an indirect simple estimate of reuse distance. Our proposal learns the collective reuse probability of the blocks in each partition on-the-fly and decides the victim caching candidates based on these probability estimates. The proposed victim cache at 64 KB capacity, on average, saves 70% LLC accesses and 12.2% execution cycles compared to a baseline 128-core system that has no private L2 cache. In contrast, a traditional 128 KB non-inclusive LRU L2 cache saves only 42% LLC accesses compared to the same baseline while performing slightly worse than the proposed 64 KB victim cache. In summary, our proposal outperforms the traditional design while halving the space investment for the per-core private L2 cache.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Mainak Chaudhuri, for his mentorship, dedication, and support throughout my graduate studies. Dr. Chaudhuri has not only been a constant source of excellent, technically sound advice, but also a joy to work with. He has always encouraged me to pursue my own ideas and not be scared of hard problems. Doing research with him has been immensely rewarding and a great learning experience.

I wish to express my gratitude to the Department of Computer Science and Engineering at IIT-Kanpur, where I realized that learning can be fun. Thanks especially to the late Dr. Sanjeev K. Aggarwal and Dr. Rajat Moona who introduced me to interesting problems in the area of computer systems, which eventually encouraged me to pursue research in computer architecture. I am thankful to Brajesh Mishra and Saurabh Malhotra for being amazing sysadmins and the continuous support they provided towards managing the compute nodes.

I have learned a lot from my fellow graduate students at IIT-Kanpur and would like to thank Arpita Korwar, Ashish Agarwal, Rohit Gurjar, Siddharth Rai, and Tejas Gandhi. Special thanks to Siddharth Rai, who provided numerous insights and feedback as well as infrastructure support needed for the work presented in this thesis.

On a more personal note, I would like to thank my friends, and my family for their unwavering support and encouragement. My parents, have been great role models, encouraging me to pursue a PhD at IIT-Kanpur. Thanks to these people I have been able to finish this dissertation while retaining my sanity, although I know some of them strongly disagree with this statement.

Contents

List of Figures

Chapter 1

Introduction

The workloads that typically run on general-purpose processors can be largely divided into two categories, namely client workloads and server workloads. The server workloads are multi-threaded or multi-process in nature where the threads or the processes carry out different types of work such as thread/process pool management, allocation of software resources (memory, buffer, etc.), compute, input/output (I/O), book-keeping/statistics collection, etc.. As a result, the server workloads demand high levels of concurrency and some guarantee on response time bound. To cater to this demand, during the past decade, high-end server-grade single-chip multiprocessors (CMPs) have sported an increasing number of high-performance independent processor cores along with a deep on-chip cache hierarchy. While the cache hierarchy and the highly-optimized core microarchitecture help achieve good single-thread performance, the large number of on-chip cores allow many threads to run concurrently achieving high levels of throughput. Such CMPs with large core counts are often referred to as many-core processors.

Figure 1.1 shows the high-level components of a typical single-chip multiprocessor. The chip has six cores numbered C0 to C5. Each core has an instruction processing pipeline optimized for single-thread performance. The pipeline interfaces with the per-core private L1 instruction (iL1) and L1 data (dL1) caches for fetching code and data. The L1 cache misses are forwarded to the per-core unified L2 cache. The L2 cache of each core connects to an interconnection network switch. The L3 cache is the last-level cache (LLC), shared by all cores, and is architected as a collection of banks. Each core has a local L3 cache bank connected to its switch, while accessing the other remote L3 cache banks requires a traversal through the interconnection network. The interconnection network may have a few switches dedicated for connecting the memory controllers. While the L2 cache misses are forwarded to the appropriate L3 cache bank derived from the physical address, the L3 cache misses are forwarded to the appropriate memory controller based on the physical address. Each memory controller may have single or multiple channels connecting usually to DDRx DRAM modules. All the switches are connected according to some topology to form an interconnection network [43]. The typical topologies include ring, mesh, and torus. The connection between two adjacent switches can be unidirectional or bidirectional. Unidirectional connections put restriction on the possible routes between two switches leading to lower overall bandwidth. Each switch contains virtual channels or queues to improve routing throughput and avoid routing deadlocks. The virtual channels can be multiplexed on a smaller number of physical networks.

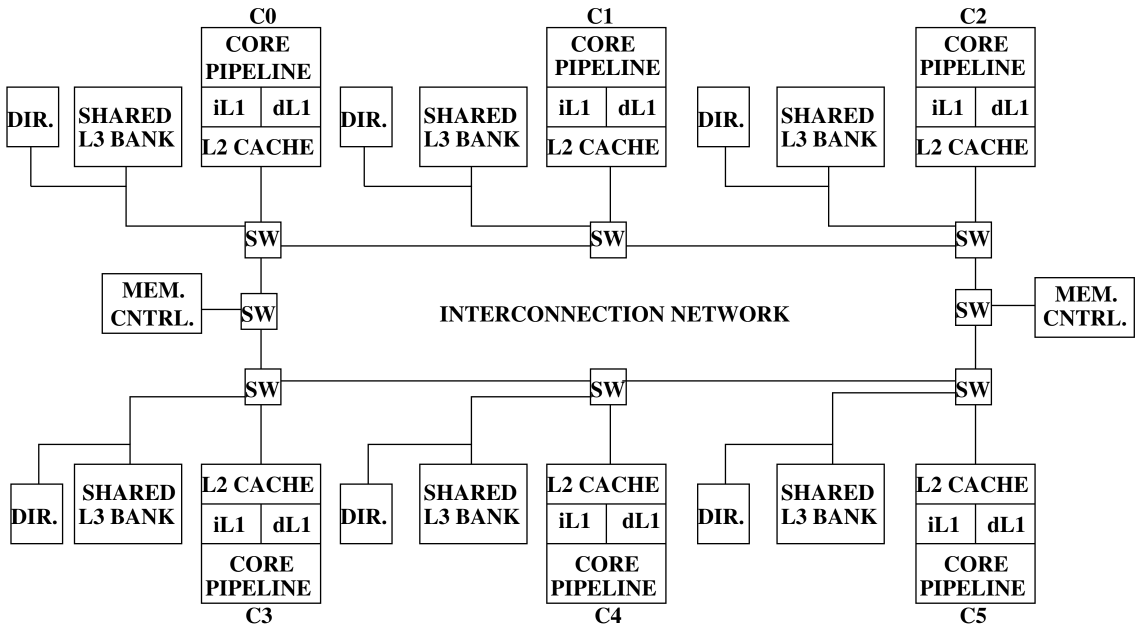

Figure 1.1: High-level architecture of a typical single-chip multiprocessor. “SW” are interconnect switches. “DIR.” is the directory storage for tracking coherence information. The network interface required for communicating with the neighboring chips is not shown.

The shared memory programming model is very popular in the multi-threaded server workloads. Its popularity primarily stems from the ease of programming enabled by the communication abstraction provided by the load/store interface. A thread can produce a new value to a shared variable through a traditional store instruction and another thread can eventually see this value by simply loading the shared variable. For this programming model to work correctly, certain ordering constraints may have to be imposed on a set of operations involving the shared variables. This is usually achieved through explicit synchronization operations in the application software. For the shared memory abstraction along with the synchronization operations to work correctly, the underlying hardware must ensure that the stores to a particular address eventually become visible to all cores in the system (known as write propagation) and that all cores in the system see the stores to a particular address in exactly the same order (known as write serialization). Write propagation and write serialization together form what is popularly known as cache coherence [25, 36, 75]. Simply put, the cache coherence hardware keeps the contents of the private caches across the cores coherent. For the cache coherence hardware to scale efficiently to a large number of cores, it is necessary to systematically avoid broadcast of any information to all cores. This requirement necessitates maintaining additional storage structures for keeping track of the location of copies of cache blocks in the entire chip. This coherence tracking information can enable the hardware to quickly locate the copies of a cache block whenever needed. For example, when a core does a store to a cache block, it may be necessary to invalidate all copies of the cache block residing in the other cores' private caches to maintain coherence. This coherence tracking storage is usually referred to as the coherence directory [14]. Further, since the directory maintains information about only a subset of the memory blocks in the system, it is referred to as the sparse directory keeping track of a sparse sampling of all the memory blocks [35, 66, 81]. The sparse directory is architected by decomposing it into slices and associating a slice with each L3 cache bank, as shown in Figure 1.1. The directory slice attached to an L3 cache bank is responsible for keeping track of the copies of all blocks mapped to that L3 cache bank. This directory slice or the associated L3 cache bank is referred to as the home directory slice or the home L3 cache bank of all these blocks. Each entry of the directory slice maintains information about one cache block. Two types of directory entry encoding have been widely used. In one type, a directory entry is represented as a vector of bits where each bit encodes either a sharer or a cluster of sharers. The width of such a bitvector entry is necessarily equal to either the maximum number of cores or the maximum number of core clusters. In the other type of encoding, a directory entry is represented as a collection of pointers where each pointer encodes the identity of a sharer. The width of such a limited pointer entry is equal to $p(\lceil \log_2 C \rceil + 1)$ where p is the number of pointers in the directory entry and C is the number of cores in the CMP. Each pointer needs a valid bit and space to encode the identity of a sharer. In addition to the sharer encoding, each directory entry maintains the coherence states of a block.

Two important problems arise as the number of cores of a cache-coherent CMP is scaled from a few tens of cores to 100+ cores. The first problem relates to the efficient support for shared memory programming model. The legacy shared memory workloads must continue to run efficiently and hence, the entire hardware substrate for cache coherence must scale gracefully [62]. However, since the coherence directory is responsible for keeping track of the copies of the blocks in the private caches, the number of directory entries must be scaled up accordingly to take into account the increasing aggregate private cache capacity with core count. An under-provisioned directory curtails the number of blocks that can be cached simultaneously in the private cache hierarchy hampering performance. Also, the average number of directory storage bits devoted to keep track of the copies of a block must be scaled up with core count because with a larger number of cores, the number of possible copies of a block inside the chip increases. This aspect usually impacts the width of a directory entry. A less than adequately sized directory entry would restrict the degree of sharing and may hamper performance. In this thesis, we study novel optimizations that target the directory width as well as the directory height for 100+ core CMPs.

The second problem relates to managing the communication traffic in the on-chip interconnection network of a large many-core processor. Given the large working sets of the server workloads, the communication traffic in the interconnect between the private cache hierarchy and the shared L3 cache banks is usually large. Additionally, as the scale of the chip increases, the average round-trip latency through the interconnect also grows. Therefore, it is important to optimize the design to reduce this traffic. The coherence directory organization and the efficiency of the private cache hierarchy together dictate the shape and volume of this traffic. While our directory optimization proposals show significant savings in the interconnect traffic, we also study the design of an efficient private cache hierarchy in the context of the server workloads. Our private cache hierarchy proposal takes into account the nature of code and data sharing between the cores and designs a space-efficient architecture that decides which blocks to keep longer in the private cache hierarchy.

In the following two sections, we outline the contributions of this thesis (Section 1.1) and its organization (Section 1.2).

1.1 Contributions

This thesis makes three specific contributions addressing the two scalability problems discussed above. The first contribution is a width-optimized novel two-level directory organization. The proposed organization attempts to optimize the average number of bits devoted to a directory entry. Our proposal exploits the well-known observation that a shared block exhibits a bi-modal pattern in the number of sharers i.e., a block is either shared by a few threads or widely shared [25, 81]. As a result, multi-threaded applications typically require two types of directory entries, namely, limited pointer entries tracking a few sharers of a block and bitvector entries tracking larger number of sharers for widely shared blocks. The primary challenge in designing a directory that has these two kinds of entries is that the exact proportion of these two types of entries varies across applications and across phases even within the same application. Recent proposals aiming to optimize the average number of bits per directory entry have organized the directory as either a static mix of these two types of entries with a pre-defined proportion or a collection of relatively short bitvector entries that can encode either a limited number of sharer pointers or a larger number of sharers hierarchically [27, 69, 85]. In either case, directory space is wasted depending on the run-time characteristics of the application. We present a directory organization that enables on-demand allocation of the two different types of directory entries dynamically. Our design maintains a pool of limited pointer entries, where each entry can also double as a segment directory entry encoding the sharers in a cluster of cores. Each entry in the primary directory array has a pointer that can either represent a sharer or point to an entry in the pool. As a result, one entry of the primary directory array is enough for tracking a private block. A moderately shared block would require a pool entry. When multiple segment directory entries are needed to encode all the sharers of a block, our pool management protocol guarantees that all these pool entries are allocated contiguously so that maintaining a pointer to the head entry is enough. We study the performance of this proposal using detailed simulation of a 128-core CMP running multi-threaded applications drawn from scientific, general-purpose, and commercial computing domains. Our Pool Directory proposal offers, on average, 5% improvement in performance and 20% savings in interconnect traffic compared to the state-of-the-art scalable coherence directory (SCD) proposal using an equally sized directory. The details of the Pool Directory design, its evaluation, and comparison with related literature are presented in Chapter 3.

The second contribution is a height-optimized directory organization. The proposed organization seeks to design a directory that has a minimal number of entries. Recent notable research efforts for optimizing the directory height have largely followed two directions. First, it has been observed that large coarse-grain contiguous memory regions are often private in server workloads [13]. Each private coarse-grain region can be tracked using a single directory entry. This optimization, however, requires support for multi-grain coherence so that fine-grain shared blocks can also be tracked by the directory [5, 10, 86]. Second, pages or blocks that are temporarily private can be identified dynamically respectively by the operating system (OS) or the hardware. The blocks thus marked private are not tracked by the directory saving directory entries. However, if and when such a block is shared at a later point, a costly non-scalable broadcast-based recovery mechanism is needed because no information about the locations of these blocks is available [24, 26].

Such broadcast-based schemes are very difficult to scale to 100+ cores. In this thesis, we design a robust minimally-sized directory that can offer adequate performance while the accompanying coherence protocol can enjoy the simplicity, scalability, and OS-independence of the traditional broadcast-free block-grain coherence protocols. We begin our exploration with a naïve design that does not have a directory at all and the location/sharers of a block are tracked by borrowing a portion of the block's L3 cache data way. Such a design, however, lengthens the critical path from two transactions to three transactions (two hops to three hops) for the blocks that experience frequent shared read accesses. This is because the L3 cache data way cannot provide the block in such cases (which the baseline can); part of the L3 cache block is corrupted with tracking information. We address this problem by architecting a tiny directory that dynamically identifies and tracks a selected subset of the blocks that experience a large volume of shared accesses. However, it is difficult to appropriately size this tiny directory, since the actual footprint of the shared blocks varies across applications and across phases even within the same application. Therefore, to make the design robust, we further augment the tiny directory proposal with an option of selectively spilling directory entries into the L3 cache space. The spill selection mechanism gives priority to the tracking information of the shared blocks that are critical and cannot be accommodated by the tiny directory. Our Tiny Directory proposal operating with under 200 KB of storage budget offers performance within a percentage of a traditional directory with 8 MB storage in a 128-core system. The Tiny Directory design outperforms the state-of-the-art by very large margins when operating with the same number of directory entries. The Tiny Directory design, its evaluation, and comparison with recent related works are discussed in Chapter 4.

The third contribution is a space-efficient private cache hierarchy design for many-core server processors. Traditionally, the private and the shared blocks are treated equally by the private cache hierarchy. We start our exploration with a single-level private cache hierarchy having a per-core private L1 cache and augment it with an intelligent L2 cache. We observe that in a single-level private cache hierarchy with per-core private L1 cache, elimination of all non-compulsory non-coherence L1 cache misses to a small subset of read-shared code and data blocks can save a large fraction of the L3 cache accesses indicating large potential for reducing the interconnect traffic in such architectures. These read-shared code and data blocks should be protected for a longer period of time in the private cache hierarchy. We architect a specialized exclusive per-core private L2 cache which serves as a victim cache for the per-core private L1 cache. The goal of the proposed victim cache is to lengthen the residency of the target subset of the read-shared blocks in the private cache hierarchy. The proposed victim cache selectively accommodates a subset of the L1 cache victims and manages the victim cache contents with specialized replacement policies. The selective victim caching and the replacement policy proposals are driven by an online partitioning of the L1 cache victims based on two distinct features, namely an estimate of sharing degree and an indirect simple estimate of reuse distance. Our proposal learns the collective reuse probability of the blocks in each partition on-the-fly and decides the victim caching candidates based on these probability estimates. The proposed victim cache at 64 KB capacity, on average, saves 70% L3 cache accesses and 12.2% execution cycles compared to a baseline 128-core system that has no private L2 cache. In contrast, a traditional 128 KB non-inclusive LRU L2 cache saves only 42% L3 cache accesses compared to the same baseline while performing slightly worse than the proposed 64 KB victim cache. In summary, our proposal outperforms the traditional design while halving the space investment for the per-core private L2 cache. The private cache hierarchy design and its detailed evaluation are presented in Chapter 5.

We summarize our contributions in the following.

- We propose a width-optimized two-level Pool Directory organization that dynamically allocates two types of directory entries from a unified pool of short-vector directory entries.

- We propose a height-optimized Tiny Directory organization that keeps track of a critical subset of shared blocks in the directory and keeps the tracking information of the rest of the blocks in the L3 cache space.

- We propose a space-efficient sharing-aware private cache hierarchy suitable for server workloads. The private cache hierarchy is designed to have a traditional L1 cache and a specialized victim L2 cache that selectively caches a subset of critical read-shared blocks.

Thesis Statement: The thesis addresses the scalability bottlenecks posed by the coherence tracking overhead and the latency/bandwidth overhead of the interconnect in a server processor with 100+ cores. The over-arching goal of the thesis is to design (i) a space-efficient coherence tracking infrastructure that can lower the coherence directory overhead while maintaining/improving performance and (ii) a space-efficient private caching architecture that can lower the overall number of trips to the shared last-level cache. The thesis demonstrates its success in achieving these goals through detailed simulation results on a 128-core chip-multiprocessor.

1.2 Thesis Organization

Chapter 2 introduces the background material on cache hierarchy, cache coherence, and directory organization relevant to our contributions. Chapters 3, 4, and 5 present the detailed proposal and evaluation of Pool Directory, Tiny Directory, and space-efficient private cache hierarchy design, respectively. Each chapter also includes a discussion of the related literature relevant to each proposal. Chapter 6 concludes the thesis with a discussion of possible future avenues of research relevant to the domain of our contributions.

Chapter 2

Background

We have introduced the readers to the basic many-core processor architecture and the notion of cache coherence in Chapter 1. In this chapter, we discuss three components of the processor microarchitecture in more detail, as these have direct relevance to the contributions of this thesis. Section 2.1 discusses the protocols for maintaining a multi-level cache hierarchy. Section 2.2 presents different types of cache coherence protocols and the basics of composing such protocols. As we have mentioned in Chapter 1, the directory storage is at the center of the scalable cache coherence hardware. We discuss some of the details of directory organization in Section 2.3.

2.1 Multi-level Cache Hierarchy

An on-chip multi-level cache hierarchy is common in today's processors. A cache hierarchy is organized as a collection of levels of caches. The cache levels get bigger and slower as they get further away from the core computing pipeline. Such a design is motivated by the fact that the most recently used code and data subset should reside close to the computing pipeline and be accessible with low latency. As a result, when an access to a cache level n fails to find the requested code or data, it is fetched from the outer levels of the cache hierarchy and usually allocated in level n, unless it can be ascertained with high confidence that the fetched block of code or data will not be accessed from the level n cache in nearfuture. In this thesis, we will assume that a block fetched to a particular level of the cache hierarchy is always allocated in that level, unless explicitly mentioned otherwise. Different cache hierarchy designs implement different types of relationships between the contents of the levels of the hierarchy and these relationships are of importance to us for this thesis. In the following discussion, we will first assume a two-level cache hierarchy with the levels denoted by L1 and L2. Next, we will extend the discussion to a three-level cache hierarchy by introducing the L3 cache.

In a two-level cache hierarchy, depending on the cache management policies, the L1 cache contents may or may not be a subset of the L2 cache contents. If the L1 cache contents are always a subset of the L2 cache contents, the L2 cache is said to be inclusive of the L1 cache. To maintain the inclusion invariant, the following two aspects of the cache management policy must be strictly observed. First, a block allocated in the L1 cache as a result of an L1 cache miss needs to be resident in the L2 cache as well. This implies that when a block is fetched from outer levels of the memory hierarchy, it is allocated in both L1 and L2 caches. Second, when a block is evicted from the L2 cache, in addition to querying the L1 cache for any up-to-date dirty copy, the L1 cache copy must also be invalidated. These invalidation messages are usually referred to as back-invalidation messages. If the L1 cache contents are not a subset of the L2 cache contents, the L2 cache is said to be non-inclusive of the L1 cache. A special case of a non-inclusive L2 cache is an exclusive L2 cache where the intersection between the L1 cache contents and the L2 cache contents is always empty. To maintain the exclusion invariant, a block allocated in the L1 cache must not be resident in the L2 cache or a block allocated in the L2 cache must not be resident in the L1 cache. This requirement implies the following. First, a block fetched from the outer levels of the memory hierarchy due to a miss in the L1 and L2 caches is allocated only in the L1 cache. Second, a block evicted from the L1 cache may be allocated in the L2 cache. Third, a block fetched to the L1 cache from the L2 cache due to an L1 cache miss is allocated in the L1 cache and invalidated from the L2 cache. Fourth, a block evicted from the L2 cache does not have to invalidate any L1 cache copy, thereby eliminating back-invalidations. The remaining spectrum of non-inclusive L2 cache designs is often referred to as non-inclusive/non-exclusive (NINE) to make it clear that it is neither inclusive nor exclusive. One popular NINE L2 cache design maintains inclusion all the time except after a block is evicted from the L2 cache. In such a design, a block fetched from outer levels of the memory hierarchy is allocated in both L1 and L2 caches. When a block is evicted from the L2 cache, its copy, if any, is not invalidated from the L1 cache. If an L1 cache eviction misses in the L2 cache, it may be allocated in the L2 cache. In this thesis, any NINE design will follow this protocol.

A two-level cache hierarchy with an exclusive L2 cache enjoys more aggregate cache capacity than one with an inclusive or a NINE L2 cache. This is because in an exclusive cache hierarchy, there is no replication of data across the levels of the cache hierarchy. Further, the back-invalidations in an inclusive design can forcefully evict live L1 cache blocks leading to loss in performance. These are usually referred to as inclusion victims. The exclusive and NINE designs do not suffer from this problem. To keep the volume of inclusion victims low, typically a 4:1 or 8:1 capacity ratio is recommended between the L2 and L1 caches of an inclusive cache hierarchy. As this capacity ratio increases, the probability that an L2 cache victim will find a copy in the L1 cache gradually drops.

The discussion presented so far can be seamlessly extended to a three-level hierarchy, which we use in this thesis. The L3 cache contents can be inclusive, exclusive, or non-inclusive/non-exclusive with respect to the union of the L1 and L2 cache contents. If the L2 cache is inclusive of the L1 cache, this union is same as the L2 cache contents. In this thesis, we will use a NINE L2 cache and a NINE L3 cache design. After eviction of a block from the L3 cache, its copy may continue to reside in the L1 cache or the L2 cache or both. This model of inclusion, exclusion, and NINE can also be extended to a setting with multiple cores. In this thesis, we will use a model where the L1 and L2 caches are private to a core, while the L3 cache is shared among all the cores. In such a NINE L3 cache design, after eviction of a block from the L3 cache, multiple copies of it may continue to reside in the L1 cache or the L2 cache or both in multiple different cores.

2.2 Cache Coherence Protocols

Cache coherence helps offer the single shared address space abstraction to the programmers in a many-core system having multiple private caches. To provide programmers the view of a coherent shared memory across all the cores, CMPs implement cache coherence protocols. These protocols keep the multiple private caches of a CMP coherent by enforcing the **S**ingle-**W**riter **M**ultiple-**R**eader (SWMR) invariant [75]. The SWMR invariant requires that for any memory location, at any given moment in time, there is either a single core that may write to it (and may also read it) or some number of cores that may read from it. A cache coherence protocol is a collection of actions that take the responsibility of propagating and serializing the stores to a particular address. The actions depend on the type of the operation e.g., a load or a read miss from a core's private cache hierarchy, a store or a write miss from a core's private cache hierarchy, or an eviction coming out of a core's private cache hierarchy. An operation completed within the confines of the private cache hierarchy of a core does not require any action from the cache coherence hardware. To implement the appropriate actions, a coherence protocol maintains the state of every cache block residing inside the private cache hierarchy. The nature of the actions also depends on the state of the block involved in the operation.

Typical cache block states are invalid (I), shared (S), clean exclusive (E), modified (M), and owned (O). A block is in invalid state with respect to a particular level of the cache hierarchy of a core if it is not present in that level of the cache hierarchy. A block is cached in shared state in a particular level of the cache hierarchy of a core if it is not modified and the core needs coherence protocol's "permission" to modify it. The core has permission to read the block. Other cores can also have copies of the block in shared state in their private caches. A block is in clean exclusive state in a particular level of the cache hierarchy of a core if it is not modified and the core has permission to read as well as modify the block. No other core can have a copy of the block in its private cache hierarchy. A block is in modified state in a particular level of the cache hierarchy of a core if parts of the block have been modified. The core has permission to read from and write to the block. A block is in owned state in a particular level of the cache hierarchy of a core if the block has been modified and the core does not have permission to modify the block. The core can read the block. No other core can have the block in E, M, or O state. Other cores can have copies of the block in shared state in their private caches. The eviction of an O state block from the private cache hierarchy of a core generates a writeback to the next level of the memory hierarchy. A coherence protocol implementation may not support all the five states. Based on the set of supported states, MESI and MOESI are the two very popular coherence protocols.

The coherence protocol actions are triggered in response to a request coming out of the private cache hierarchy of a core. To trigger the appropriate action, knowledge about the current state of the requested block in the private caches of other cores is necessary. There are two possible techniques to gather this information. In the first technique, the request is broadcast to all cores. The private caches of each core "snoop" the request and act depending on the state of the requested block in its private cache hierarchy. These broadcast-based snoopy protocol implementations demand significant interconnect bandwidth and usually considered impractical beyond a few tens of cores.

In the second technique, as mentioned in Chapter 1, a dedicated piece of storage, known as the coherence directory, is maintained for tracking locations and state of each block resident in the private caches of the cores. The locations encode the id's of the cores that have copies of the block in their private caches. Depending on the requested block address, the request is first sent to the home L3 cache bank where it consults the home directory slice (the concept home was introduced in Chapter 1). The directory offers the current locations of the copies of the block and indicates the coherence state of the block. The coherence state can be S, M, or O depending on the supported set of states. A coherence state of S indicates that the block is in shared state and the rest of the directory entry provides the identities of the sharers (there could be only one sharer also). A coherence state of M indicates that the block is in modified state and the rest of the directory entry encodes the identity of the core where the modified block can be found. A coherence state of O indicates that the block is in O state in the private cache hierarchy of a core. In this case, the rest of the directory entry encodes the identity of the owner core and the identities of the other sharers of the block. A coherence state of I is unnecessary because a block that is not resident in the private cache hierarchy of any core is not tracked in the directory. The directory is a tagged structure and the tags of only those blocks which are cached in the private cache hierarchy of the cores are found in the directory. A coherence state of E is not maintained in the directory and a block in E state in some core's private cache hierarchy is tracked in M state in the directory. This is because the core having the block in E state can silently modify the block and take the block to M state without notifying the home directory slice.

Once the locations and the state of a requested block are obtained from the directory, point-to-point unicast messages are sent to only the cores having copies of the requested block. Only these cores act upon the forwarded request. Since a request is always forwarded to the home directory slice first before any coherence action is triggered, the directory also serves as a serialization point for multiple racing requests to a block. The directory handles these requests one at a time typically in the order of arrival (although this order has no influence on the correctness) and resolves the races with additional transient coherence states. Naturally, the directory-based coherence protocol implementations demand much less interconnect bandwidth than their broadcast-based snoopy counterparts and hold promise to scale to 100+ cores. In this thesis, we will consider only directory-based coherence protocols. We discuss the basic set of actions of such protocols in more detail in the next section.

2.2.1 Directory-based Cache Coherence Protocols

As already mentioned, the exact set of coherence protocol actions depends on the request type and the current state of the requested block. In the following, we discuss the different cases for a MESI cache coherence protocol, which we use in this thesis. We assume a NINE L3 cache design. We do not assume any point-to-point message ordering between pairs of source and destination in the interconnection network. This is because different message types (e.g., request, response, intervention) for the same block between the same pair of source and destination may travel along different virtual networks inside a switch and may not maintain any ordering along the way in the most general unrestricted design of virtual network scheduling algorithms, which usually carry out an online maximum bipartite matching between the input and output ports of a switch. This is true even if all virtual networks multiplex on a single physical network. Mapping different types of messages on different virtual networks is necessary to avoid protocol deadlocks [18]. Even within the same virtual network, it is possible to reorder the messages for improving program performance e.g., it may make sense to prioritize processor miss requests over writebacks in the request network. Also, two different messages in the same virtual network between the same source and destination pair may travel along two different paths if adaptive or hot-potato routing is supported. Overall, the most general coherence protocols do not assume any ordering between network messages.

Read/Load miss

Suppose core C suffers from a read miss in its private cache hierarchy for a block B. The request is sent to the home L3 cache bank of B and the home directory slice is consulted in parallel with reading the L3 cache tags. The following situations may arise.

B **is not in L3 cache and not tracked in directory:** The request is forwarded to the appropriate memory controller which responds with the block after reading from DDRx DRAM. The block is allocated in the home L3 cache bank and forwarded to the private cache hierarchy of C for allocation in E state if it is a data block or in S state if it is a code block. The home directory slice allocates an entry for tracking B, marks the coherence state of the block as M or S depending on whether the block is data or code respectively, and records the identity of C in the entry.

B **is in L3 cache, but not tracked in directory:** The block is read from the L3 cache and sent to the private cache hierarchy of C for allocation in E state if it is a data block or in S state if it is a code block. The home directory slice allocates an entry for tracking B, marks the coherence state of the block as M or S depending on whether the block is data or code respectively, and records the identity of C in the entry. This is known as a two-hop transaction because the critical path involves two messages. This is shown in Figure 2.1.

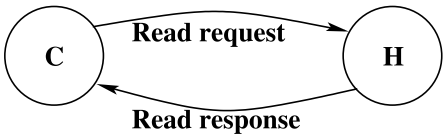

Figure 2.1: A two-hop read transaction. H represents the home L3 cache bank and its associated directory slice.

B **is in L3 cache and tracked in directory:** The actions depend on the coherence state of the block recorded in the directory entry. In a MESI protocol, the coherence state can be either S or M. If the state is S, the block is read from the L3 cache and sent to the private cache hierarchy of C for allocation in S state. The identity of core C is added to the list of sharers in the directory entry. This is a two-hop transaction. On the other hand, if the coherence state is M with the identity of core C' recorded in the directory entry, the request is forwarded to core C' and the directory entry is marked busy which is a transient coherence state. This forwarded message is usually referred to as an intervention message. Core C' directly sends the requested block to core C and also sends a message (typically known as sharing writeback [25]) to the home directory slice to change the state of the directory entry to S and mark both C and C' as sharers in the directory entry. The sharing writeback message also carries a copy of the block to update the copy in the L3 cache bank so that subsequent requests for this block can be responded to by the L3 cache bank itself. The critical path involves three messages and the transaction is referred to as a three-hop transaction. This is shown in Figure 2.2. It may happen that when the forwarded intervention message arrives at core C', the requested block has already been evicted from the private cache hierarchy of core C'. To handle such late intervention races, each evicted block is allocated in an eviction buffer local to the private cache hierarchy of the core evicting the block. The block stays in the eviction buffer until the eviction is acknowledged by the home directory slice. Every eviction message received by the home directory slice is acknowledged with one of the two acknowledgment message types sent back to the evicting core. Two acknowledgment types are needed to differentiate the situation where the home directory has forwarded an intervention from the situation where there is no such in-flight intervention message.

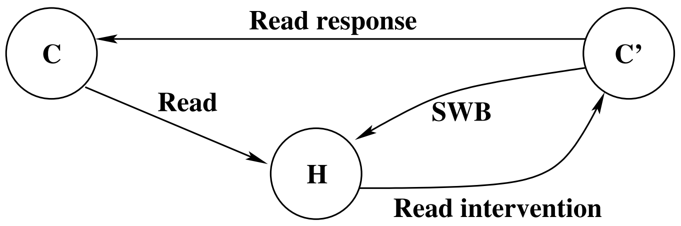

Figure 2.2: A three-hop read transaction. *H* represents the home L3 cache bank and its associated directory slice. SWB is the sharing writeback message.

***B* is not in L3 cache, but is tracked in directory:** This situation can arise in a NINE L3 cache design where the L3 cache has evicted the block, but some cores still have the block in their private caches. If the coherence state of the block is M in the directory entry, an intervention message is forwarded to the core having the block and the protocol actions are similar to a regular intervention as discussed above. The only additional action is to allocate the block in the L3 cache bank when the sharing writeback message arrives. If the coherence state of the block is S in the directory entry, one of the sharers is elected to provide the block and an intervention message is forwarded to the elected sharer. The rest of the flow is same as a regular intervention discussed above, except that a copy of the block is allocated in the L3 cache so that subsequent requests can be quickly served from the L3 cache bank itself using a two-hop transaction. The late intervention races are handled with the help of the eviction buffer and eviction acknowledgment messages.

An alternative to dynamically electing a sharer as the source of the block is to maintain a "forwarding" core all the time in the directory entry. Such protocols are referred to as MESIF protocols, where there is always a core having the block in F state. The primary complication of such a protocol is that when the F state core evicts the block from its private cache hierarchy, a new forwarding core must be elected from among the existing sharers of the block by the home directory slice and notified. This requires complex handling of races where the newly elected core may also evict the block while the election is in progress in the home directory slice. We do not explore MESIF protocols in this thesis. We find that a reasonably small eviction buffer per core is sufficient for our implementation.

Write/Store Miss

Suppose core C suffers from a store miss in its private cache hierarchy for a block B. A store miss can take two forms depending on the state of the block B in the private cache hierarchy of C. If the block is not present in C, a read-exclusive miss is generated. On the other hand, if the block is present in C is S state, an upgrade miss is generated. The sole purpose of the upgrade miss is to seek permission from the cache coherence layer to modify the block B. In any case, the miss request is sent to the home L3 cache bank of B and the home directory slice is consulted in parallel with reading the L3 cache tags. The following situations may arise.

B is not in L3 cache and not tracked in directory: In this case, the miss request must be read-exclusive. The request is forwarded to the appropriate memory controller which responds with the block after reading from DDRx DRAM. The block is allocated in the home L3 cache bank and forwarded to the private cache hierarchy of C. The home directory slice allocates an entry for tracking B, marks the coherence state of the block as M, and records the identity of C in the entry.

B is in L3 cache, but not tracked in directory: In this case also, the miss request must be read-exclusive. The block is read from the L3 cache and sent to the private cache hierarchy of C. The home directory slice allocates an entry for tracking B, marks the coherence state of the block as M, and records the identity of C in the entry.

B is in L3 cache and tracked in directory: The actions depend on the coherence state of the block recorded in the directory entry and the request type. Let us first consider the case of a read-exclusive request. If the coherence state in the directory entry is S, the block is read from the L3 cache and sent to the private cache hierarchy of C along with the number of sharers. An invalidation message is sent to each sharer. The identity of core C is recorded in the directory entry and the state of the directory entry becomes M. A sharer, on receiving an invalidation message, invalidates its cached copy of block B and sends an invalidation acknowledgment to the requester core C. When the number of invalidation acknowledgments received by C matches the sharer count, the store miss completes. The involved messages are shown in Figure 2.3. It is possible to let the requesting core commit the store operation even before all invalidation acknowledgments are collected. This is known as eager-exclusive response [3]. Use of eager-exclusive responses eliminates the invalidations and their acknowledgments from the critical path of the request, but leads to relaxed memory consistency models that can significantly deviate from the intuitive sequential consistency model [3]. This thesis implements sequential consistency model only and, as a result, cannot use eager-exclusive responses.

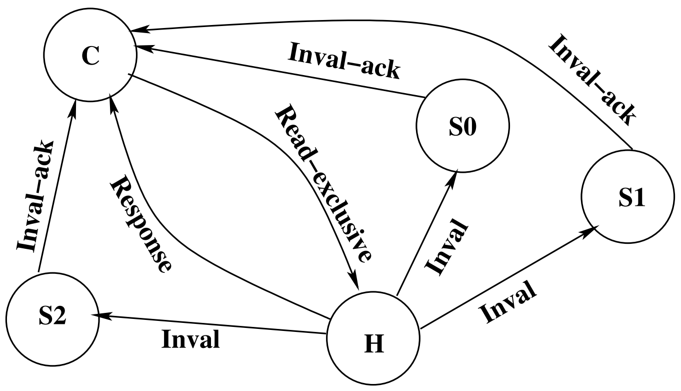

Figure 2.3: A read-exclusive request requiring invalidations (Invals). The sharers are denoted S0, S1, S2.

If the coherence state is M with the identity of core C' recorded in the directory entry, the request is forwarded to core C' and the directory entry is marked busy which is a transient coherence state. Core C' directly sends the requested block to core C, invalidates its copy of block B, and also sends an ownership transfer message to the home directory slice to change the state of the directory entry to M and record the identity of C in the directory entry. This is shown in Figure 2.4. Unlike the sharing writeback message, the ownership transfer message does not carry any data. The late intervention races are handled with the help of the eviction buffer and eviction acknowledgments, as usual.

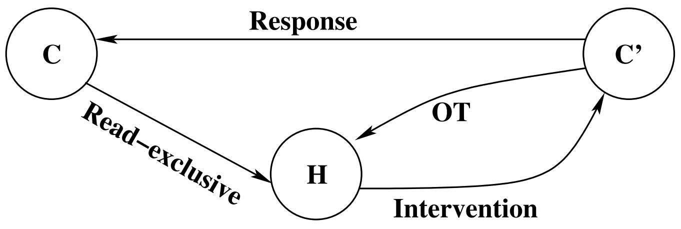

Figure 2.4: A three-hop read-exclusive transaction. OT is the ownership transfer message.

Next, let us consider the case of an upgrade request. If the coherence state in the directory entry is S, an upgrade acknowledgment (a dataless message) is sent to the private cache hierarchy of C along with the number of sharers minus one (the requesting core C is also a sharer and must be excluded from this count). The invalidation messages and collection of invalidation acknowledgments are exactly same as discussed above, except that no invalidation is sent to the requesting core C, which is also a sharer. If the coherence state in the directory entry is M with the identity of core C' recorded in the directory entry, this means that core C' must have invalidated the copy of the block in the private cache hierarchy of C. The request is changed to read-exclusive and forwarded to C'. The directory entry is marked busy which is a transient coherence state. The handling of the intervention message and the ownership transfer message is exactly same as already discussed.

B is not in L3 cache, but is tracked in directory: If the coherence state of the block is M in the directory entry, an intervention message is forwarded to the core having the block and the protocol actions are similar to what is already discussed regarding handling of intervention messages. If the coherence state of the block is S in the directory entry, the protocol actions depend on the request type. For a read-exclusive request, one of the sharers is elected to provide the block and an intervention message is forwarded to the elected sharer. The rest of the flow is already discussed. For an upgrade request, an upgrade acknowledgment is sent to the private cache hierarchy of C along with the number of sharers minus one. The invalidation messages and collection of invalidation acknowledgments are exactly same as already discussed.

Eviction from Private Cache Hierarchy

All evictions from the private cache hierarchy are allocated in the eviction buffer of the evicting core. Only M state evictions carry the evicted block to the home L3 cache bank (traditional writeback messages). The E and S state evictions generate dataless messages to the L3 cache bank. Sending S state eviction notices allows us to keep the directory entry up-to-date and avoids future unnecessary invalidations and invalidation acknowledgments [62]. On receiving an eviction message with data, the home L3 cache bank updates its copy (if it has a copy). In all cases, the home directory slice is consulted and the directory entry is updated to reflect the eviction. The home L3 cache bank sends an eviction acknowledgment message back to the core which sent the eviction message. If the eviction acknowledgment message indicates that there is an in-flight forwarded intervention message for the evicted block, the evicting core waits for the intervention and does not de-allocate the evicted block from the eviction buffer until the intervention is received. On the other hand, if the eviction acknowledgment message indicates that there is no in-flight intervention, the evicting core de-allocates the evicted block from the eviction buffer on receiving the eviction acknowledgment message.

If a read or a read-exclusive miss request from a core finds the requested block in the local eviction buffer, the request is delayed until the eviction buffer entry is de-allocated. This is necessary to avoid a race where the read or the read-exclusive miss request reaches the home L3 cache bank before the eviction message and reads a stale copy of the block. While this race is relevant to M state evictions only, the race involving the directory entry update is relevant to all evictions. In the latter race, the read or read-exclusive message from a core C races past the eviction message from the same core for the same block, reaches the home L3 cache bank, and updates the directory entry to record the identity of core C. The eviction message later removes the identity of core C from the directory entry, which leaves the directory entry in a wrong state. Allocating all evictions in the eviction buffer avoids these races.

Eviction of Directory Entry

A directory entry may have to be evicted due to shortage of directory storage capacity or conflict in directory entry allocation. In such an event, if the coherence state of the block tracked by the evicted directory entry is S, all sharers are sent invalidations; if the coherence state is M, the core having the block is sent an intervention to retrieve and invalidate the block from the core's private cache hierarchy and the copy in the home L3 cache bank is updated. In summary, early eviction of directory entries may hamper performance by prematurely invalidating blocks from the private cache hierarchy of the cores. An under-provisioned directory limits the maximum number of blocks that can be simultaneously resident in the private cache hierarchy of the cores. This brings us to the interesting topic of organizing and sizing the directory. We discuss it in the next section.

2.3 Directory Organization

The coherence directory keeps track of the locations and the coherence state of each block resident in the private caches of the cores. As mentioned in Chapter 1, this structure is referred to as a sparse directory. In the following discussion, we will first introduce the readers to the organization of a monolithic sparse directory. Next, we will decompose it to construct the slices.

The sparse directory is organized as a set-associative cache with a tag array and a data array. Each entry of the data array has two components, namely the coherence state and encoding of the locations of the copies of the block being tracked by this entry. Ideally, the sparse directory should be organized and sized in such a way that it can track all the blocks in the private caches of the cores. For example, in a two-level private cache hierarchy having an inclusive L2 cache, the number of sets in the sparse directory should be equal to the number of sets in the L2 cache and the associativity of the sparse directory should be equal to the associativity of the L2 cache multiplied by the number of cores. Such an organization of the sparse directory guarantees that every L2 cache block has a corresponding directory entry. Since the L2 cache is inclusive of the L1 cache, this organization also captures all blocks resident in the L1 cache. If the L2 cache is non-inclusive/non-exclusive or exclusive, the ideal sparse directory organization becomes complex. Nonetheless, the ideal sparse directory organization does not scale favorably with core count. For 100+ cores, the associativity would be impractically large. As a result, a practically implementable sparse directory cannot guarantee a one-to-one correspondence between the directory entries and private cache blocks. Hence, there can be conflicts in the directory between blocks which do not conflict in the private cache hierarchy. To keep the conflict volume low, the sparse directory is usually over-provisioned. In this thesis, we denote the number of entries in the sparse directory as $R\times$ when the sparse directory has R times as many entries as the number of L2 cache blocks aggregated across all cores. An over-provisioned directory would have R more than one, while an under-provisioned sparse directory would have R less than one. For example, in a 128-core processor having a per-core 128 KB L2 cache with 64-byte block size, a $2\times$ sparse directory would have 512K entries. For the same configuration, a $\frac{1}{16}\times$ sparse directory would have 16K entries. These entries are organized as a set-associative cache. The sets are distributed equally across the L3 cache banks to construct the slices. For example, if a sparse directory has 512K entries organized into eight ways, the total number of sets in the directory is 64K. If there are 128 L3 cache banks, each directory slice would have 512 sets, each being eight-way associative.

We have already discussed the coherence states for a MESI protocol. We discuss two basic techniques for encoding the locations of the copies of a block in a directory entry. The other encodings are usually a combination of these two basic techniques. In the first technique, a bitvector is used for encoding the location of a copy. For example, if there are n cores, a full-map bitvector would have n bits. A copy of the block present in the private cache hierarchy of core i is indicated by setting bit i of the bitvector. If we assume the per-core private cache capacity to be a constant, a sparse directory employing full-map bitvector entries has a storage overhead growing as $O(n^2)$ with the core count n. It is also possible to use a bitvector of shorter length. This is usually referred to as a coarse-vector representation [25] and a bit represents a cluster of cores. For example, a bit in a vector of length $n/4$ would represent a cluster of four cores. If any core in the cluster has a copy of the block, the corresponding bit in the vector would be set. The coarse-vector representation becomes an attractive option when the full-map bitvector's overhead becomes prohibitive.

In the second technique, the locations are encoded using a set of pointers. If there are n cores in the system, each location pointer requires $(\lceil\log_2 n\rceil + 1)$ bits to encode the identity of a core and a valid bit. If the number of bits devoted to encoding the locations is N, a directory entry can encode $\lfloor N/(\lceil \log_2 n \rceil + 1) \rfloor$ pointers. This is known as a limited pointer encoding and is useful for encoding only a small number of sharers. This encoding is very popular when N is smaller than n i.e., a full-map bitvector is not affordable. In this situation, a limited pointer encoding is used initially until the number of sharers becomes more than $\lfloor N/(\lceil \log_2 n \rceil + 1) \rfloor$. When the sharer count exceeds this limit, the encoding may switch to a coarse-vector representation. Since the coarse-vector representation is an approximate one, it can lead to extra useless invalidation messages on store misses. Therefore, the use of limited pointer representation for the blocks with a small number of sharers eliminates this problem; only the widely shared read/write blocks suffer from extra invalidation traffic arising from inexact coarse-vector representation.

Chapter 3

Pool Directory

The sparse directory maintains information about the blocks resident in the private cache hierarchy of each processing core in a CMP. As the on-chip cores grow in number, the design of the sparse directory needs to be space-efficient so that these systems can continue to support the shared memory abstraction with acceptable directory storage budget [62]. The sparse directory organization has become popular due to its simplicity and space-efficiency. While the sparse directory provides an attractive starting point for optimizing the number of directory entries, the width of a sparse directory entry still needs to scale linearly with the core count if the entry is organized as a full-map bitvector. The full-map bitvector organization is the simplest possible sharer encoding from the viewpoint of the hardware required to manipulate the directory information. In this chapter, we focus on the problem of optimizing the average number of bits devoted to a sparse directory entry. We begin with a motivating study to quantify the utilization of a full-map sparse directory entry in a 128-core CMP.

The attempts to optimize the width of a sparse directory entry exploit the observation that at a given point in time, not all blocks need a full-map bitvector. The degree of sharing varies across blocks [82] and over time within the same application. This is usually reflected in the utilization of the sparse directory. Figure 3.1 presents one such study on our simulated model of a 128-core CMP. The left panel of Figure 3.1 shows the percentage of the allocated directory entries that experience a maximum of k sharers where k falls in one of four disjoint sharer count bins: 2 to 4, 5 to 8, 9 to 16, and 17 to 128 (end-points inclusive). These data are collected on a 128-core system for fourteen multi-threaded applications spanning the PARSEC suite [12], the SPLASH-2 suite [83], SPEC JBB, SPEC Web running on the Apache server, TPC running on the MySQL server, and the SPEC JVM suite.[1] For these measurements we use a 2$\times$ sparse directory so that premature directory evictions do not hamper the number of blocks that can be resident simultaneously in the private cache hierarchy and can be shared. These data show that, on average, only 10% directory entries observe any sharing, while the rest of the allocated directory entries track only private blocks. Most of the sharing instances are limited to at most four sharers, while only two applications (swaptions and barnes) show noticeable number of directory entries experiencing more than sixteen sharers. These data indicate that, on average, a large number of bits in a full-map directory entry is wasted due to lack of high volume of sharing. This is further confirmed in the right panel of Figure 3.1 which shows the average percentage of set bits in an allocated directory entry during its residency in the sparse directory. Across the fourteen applications, on average, only 2.4% bits in a full-map directory entry are ever set. These data clearly indicate the importance of optimizing the directory entry width.

The proposals aiming to optimize the sparse directory width while preserving the preciseness of sharing information and not resorting to broadcast, overflow-induced early invalidations, or software solutions organize the directory in one of the following three forms: 1) a statically designed mix of different types of entries [27], 2) a hierarchical organization of the sharing bitvector by decomposing the system into a hierarchy of clusters [69], and 3) dynamic allocation of tracking information [73, 74, 85]. Based on the observation that the private blocks are often more in number than the shared blocks, it has been proposed that the sparse directory sets be designed to have a static mix of pointer and bitvector ways [27]. The pointer ways track private blocks and have width that grows logarithmically with core count. The bitvector ways track shared blocks and their width grows linearly with core count. Due to static partitioning of each set into two types of

[1] Details of our simulation environment are discussed in Section 3.3.

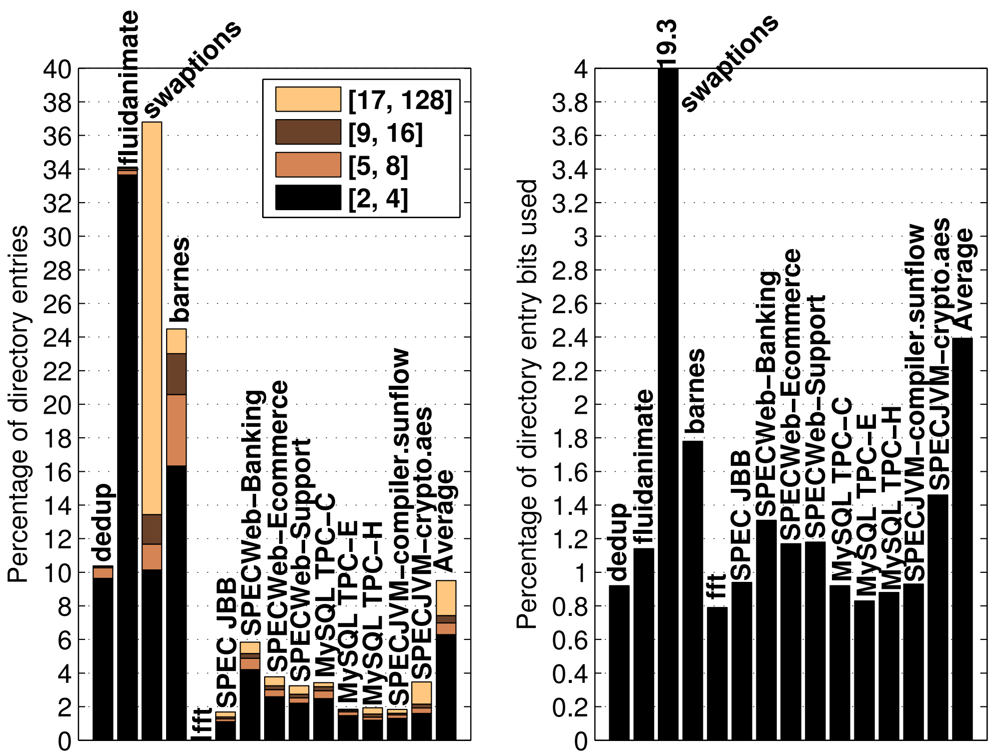

Figure 3.1: Left panel: Distribution of maximum sharer count averaged over all allocated directory entries in a 2× sparse directory. Right panel: Percentage of directory entry bits set averaged over all allocated directory entries in a 2× full-map sparse directory.

entries (e.g., six pointer ways and two bitvector ways in an eight-way set-associative sparse directory), such a design cannot react optimally to the changing sharing degree. We will refer to this design as the Hybrid directory organization.

The scalable coherence directory (SCD) is the state-of-the-art hierarchical sparse directory organization [69]. The SCD proposal represents the core count as a product of two integers (ideally equal) and the larger integer decides the width of the directory. This organization treats a system with pq cores as p clusters of q cores each, where q is bigger than or equal to p. Each directory entry is q bits wide. Multiple directory entries are used to encode the sharers in a hierarchical fashion. Each bit in the top-level or root entry represents a cluster and in the worst case $p+1$ directory entries would be needed to encode all the sharers. The non-root entries will be referred to as the leaf entries. In any case, at least two directory entries are required and the best case arises when all the sharers are confined to a single cluster. Additionally, each directory entry can encode a certain number of sharer pointers ($\lfloor q/(\lceil \log_2(pq) \rceil + 1) \rfloor$) and the coherence tracking format of a block switches to the hierarchical one only after the number of sharers of the block exceeds this limited pointer count that each directory entry can accommodate. While this proposal is able to achieve a square root factor reduction in the directory width, multiple directory entries have to be invested to encode the sharers of a block. This increases the pressure on the directory as the degree of sharing and the number of shared blocks increase. Since each involved directory entry carries the tag of the shared block, the tracking overhead per shared block increases significantly in the worst case. For example, in a 1024-core system having a 32-bit wide hierarchical directory, 33 directory entries (one root and 32 leaves) would be needed in the worst case to encode the sharers of a block. If we assume a 24-bit tag, the worst case tracking overhead per shared block is $33 \times (24+32)$ bits, which far exceeds the tracking overhead for a full-map bitvector (1024+24 bits). The relative wastage increases in smaller core-count systems. The observed level of inefficiency, however, depends on the actual sharing pattern of the application. An additional inefficiency arises due to a large volume of private blocks. Each of these blocks needs just one pointer and wastes the remaining pointers in a directory entry.

Dynamic allocation of tracking information has been proposed in two different forms. The dynamic pointer allocation scheme assigns a new sharer pointer entry to a block as and when a new sharer is added [73, 74]. Searching the list of sharers for a specific sharer or walking the sharer list for sending out invalidations can be costly. A recent proposal carries out dynamic assignment of a full-map bitevector to a sparse directory tag whenever the block corresponding to the tag gets shared by at least two cores [85]. For tracking private blocks, the proposal uses a single owner pointer attached to the tag. While this proposal is able to eliminate the directory bit wastage for tracking private blocks, the space inefficiency is still significant for tracking shared blocks. For example, a block with k sharers would waste $n-k$ bits of the full-map bitvector in an n-core system. This design will be referred to as the Select directory conveying the fact that only a selected subset of the sparse directory tags gets dynamically attached to full-map bitvectors as the need arises.

In this chapter, we introduce Pool directory, a novel sparse directory organization that aims at optimizing the average number of bits per directory entry. Our proposal organizes the entire directory as a two-level structure. The first level is the sparse directory and the second level is a separate direct-mapped pool of short vector entries. The first level directory will be referred to as the sparse directory and the second level will be referred to as the pool. Each sparse directory entry contains a single pointer. Each pool entry can store either a few sharer pointers or a sharer vector encoding the sharers in a cluster of cores. To encode all the sharers of a block, multiple pool entries may be needed and these are allocated dynamically as and when the need arises. Our pool management protocol ensures that all these pool entries remain contiguous so that it is sufficient to maintain a pointer to the head entry. The pointer in each sparse directory entry can either encode a sharer (particularly useful for tracking private blocks) or point to a pool entry. For example, for a block with a small number of sharers, a single pool entry would suffice to track the block and in this case, the pointer in the sparse directory entry points to the pool entry. On the other hand, for a widely shared block, a contiguous block of pool entries would be needed to track the block and in this case, the pointer in the sparse directory entry points to the first pool entry of the block of pool entries.

Our proposal enjoys four distinct advantages compared to the state-of-the-art hierarchical representation. First, the private blocks never contend for pool entries and can be encoded in the sparse directory array. Second, exactly one tag in the sparse directory array is allocated for tracking a block. Each individual pool entry does not need a separate tag. Third, there is no need to maintain a root entry in the pool. At most p pool entries are needed to encode p clusters of q cores each assuming q-bit wide pool entries. Fourth, the sharers of a block can be encoded by optimally mixing the limited pointer and the sharer vector representations in the pool entries allocated to the block, thereby offering significant flexibility in encoding the sharers in a space-efficient manner. The directory storage is dynamically allocated to track sharers of a block as and when needed by allocating additional pool entries. Additionally, the decoupled two-level organization of the directory allows us to independently size the number of entries in the sparse directory table and the pool table. We review the contributions related to optimization of directory entry width in Section 3.1. We present the detailed design of the Pool directory in Section 3.2. Our simulation results (Sections 3.3 and 3.4) on a 128-core system show that our proposal outperforms the state-of-the-art dynamic hierarchical scalable coherence directory (SCD) [69] by 5% while reducing the interconnect traffic by 20% when using a $\frac{1}{16}\times$ sparse directory. Additionally, our proposal performs within 2.4% of a full-map organization while requiring only one-third of the directory storage of a full-map organization.

3.1 Related Work

A large body of research on coherence directory store optimization has followed the first proposal that introduced a bitvector as the directory element [14]. The early proposals focused on the distributed shared memory multiprocessor architectures. The proposals for optimizing the width of the directory include limited pointer schemes with broadcast on overflow (Dir_iB), limited pointer schemes with invalidation on overflow (Dir_iNB), limited pointer schemes with software handling on overflow (LimitLESS directory), coarse-vector schemes where each bit in a vector tracks a cluster of sharers requiring a broadcast invalidation within a sharing cluster on a write, and use of gray codes for achieving better compression in the bitvector [4, 15, 35, 64]. The scalable coherent interface standard forms a doubly linked list of sharers with the help of pointers attached to each private cache block, while the directory maintains a pointer to the head of the list [42]. Although such a distributed linked list scheme is more scalable than a bitvector scheme in terms of storage, the protocol operations are significantly more complex than a bitvector protocol. Some of the proposed limited pointer schemes use a distributed linked list or a distributed tree to track the overflown sharers with one of the pointers in the directory entry serving as the head of the list or the root of the tree [16, 19]. A number of proposals organize the directory in a hierarchical tree or multi-level clusters [34, 61, 65, 80]. Although the directory organizations in these proposals achieve low overhead, the hierarchical coherence protocol makes the overall design complex. Tree-based compression of tracking information and a two-level directory architecture where the second-level directory maintains imprecise compressed information have also been proposed [1, 2]. The segment directory design partitions the system into a few clusters and tracks the sharers in a limited number of clusters [21]. On an overflow, one of the well-known overflow solutions is invoked.

More recent proposals have focused on sparse directory space optimization for CMPs. One proposal mixes two types of directory entries, namely pointers and bitvectors to design a sparse directory set [27] (will be referred to as the Hybrid directory scheme). Another proposal designs a scalable coherence directory (SCD) by representing sharers in two-level hierarchical bitvectors [69]. These two proposals have already been discussed. A directory architecture that eliminates the duplicate tag overhead of the directory by maintaining an array of Bloom filters for answering set membership queries about the sharers has been proposed [87]. This design suffers from scalability issues, since each directory access involves looking up C Bloom filters, C being the number of cores. Proposals that track a small set of sharing patterns and link each active directory entry to a sharing pattern have been proposed [89, 90]. Limited pointer representations that use one pointer for counting the sharers on overflow have been explored [55]. This count is later used to limit the number of acknowledgment messages needed on a broadcast invalidation. A recent proposal has used multi-level memristors to compress the size of the directory entries [88]. In contrast to these, our proposal presents a design for dynamically allocating directory entries while leaving the cache coherence protocol unchanged.

3.2 Pool Directory

We discuss the detailed design of the Pool directory in the following. Section 3.2.1 presents the architecture of the Pool directory and Section 3.2.2 discusses the implementation of the various operations supported by the Pool directory. In this discussion, we assume a traditional MESI cache coherence protocol [56]. The most important actions of such a protocol were discussed in Chapter 2.

3.2.1 Directory Organization

The Pool directory architecture consists of two structures, namely, the sparse directory and a pool of short sharer vectors, as shown in Figure 3.2. The sparse directory is a set-associative array with each entry having a tag and tracking/state information of the block corresponding to the tag. The major part of the tracking information is taken up by a pointer (P), which can either encode a sharer or point to an entry in the sharer vector pool. If the number of cores in the system is C and the number of entries in the sharer vector pool is N, each pointer needs to be $\lceil \log_2(\max(C, N)) \rceil$ bits wide. The single sharer (S) bit in a sparse directory entry is set when the corresponding block has a single sharer (i.e. private), which is directly encoded in the pointer P.

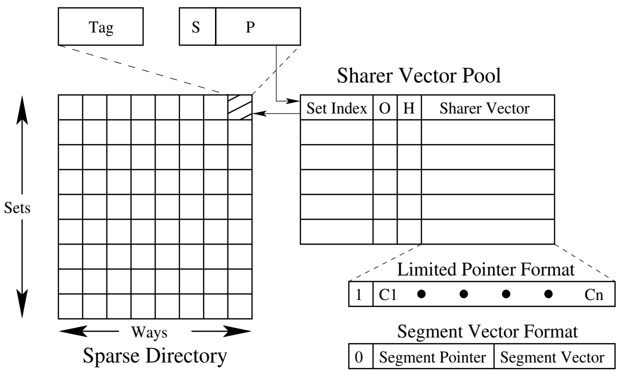

Figure 3.2: General structure of the Pool directory.

The sharer vector pool is a tagless direct-mapped array with each entry having a sharer vector. The sparse directory entries for blocks having more than one sharer utilize a collection of consecutive pool entries to encode the sharers. Each such collection of pool entries can be logically seen as a linked list with a head pool entry and a tail pool entry. A sparse directory entry that needs pool entries has its single sharer (S) bit reset and the pointer (P) points to the head entry of the collection of pool entries being used. Each pool entry has a head (H) bit indicating if the entry is a head entry for some sparse directory entry. Each pool entry also contains an occupied (O) bit and the set index of the sparse directory entry it is associated with. The occupied bit is set for a pool entry in use. We will discuss the utility of the occupied bit, the head bit, and the set index field in Section 3.2.2. We note that the private blocks do not need any pool entry.

Each sharer vector can encode sharers, either in *limited pointer* format or in *segment vector* format. As shown in Figure 3.2, the first bit of each sharer vector identifies the encoding format being used by the vector. When the first bit is set, the sharer vector is encoded using the limited pointer format and when the first bit is reset, the sharer vector is encoded using the segment vector format. The limited pointer format is useful for efficiently encoding a small number of sharers. In the limited pointer format, the sharer vector is organized as an array of pointers and their valid bits. Each pointer can independently point to a sharer of the corresponding sparse directory entry. Each such pointer needs $\lceil \log_2(C) \rceil + 1$ bits, where C is the number of cores. The segment vector format was introduced in [21] for efficiently encoding a cluster of sharers in a segment directory entry. In the segment vector format, a sharer vector consists of a segment vector and a segment pointer. The segment vector is a K-bit wide segment of a full-map vector. The segment pointer is a $\lceil \log_2(C/K) \rceil$-bit field maintaining the position of the segment vector within the full-map vector effectively recording the id of the cluster the segment represents. The limited pointer format and the segment vector format permit each sharer vector to independently encode sharers. This allows a sparse directory entry to simultaneously utilize both the encoding formats to achieve greater storage efficiency. In the worst case, $\lceil C/K \rceil$ pool entries would be required for encoding a full-map vector assuming K-bit wide pool entries.

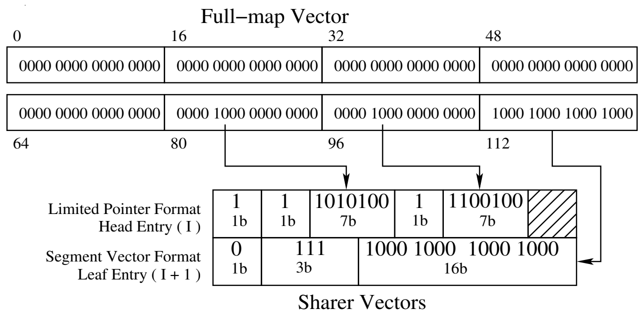

Figure 3.3: Pool entries with twenty-bit wide sharer vector in a 128-core system.

Figure 3.3 illustrates different sharer vector encoding formats for pool entries with twenty-bit wide sharer vector in a 128-core system. Each sharer vector can encode a maximum of two sharers in the limited pointer format, as a valid bit and a seven-bit wide pointer are required for encoding a sharer. When using the segment vector format, sixteen sharers can be encoded in a sixteen-bit wide segment vector. A three-bit wide segment pointer is maintained for encoding eight distinct sixteen-bit wide segments of the full-map vector. In the worst case, eight pool entries would be required to encode a full-map vector. In Figure 3.3, a sparse directory entry for a block is using two consecutive pool entries I and $I + 1$ for encoding six sharers of the block. The head entry (I) is using the limited pointer format for encoding two sharers from the 5th and the 6th segments of the full-map vector. The tail entry ($I + 1$) is using the segment vector format for encoding all the sharers from the 7th segment of the full-map vector.

3.2.2 Directory Operations

The coherence directory is looked up in parallel with every L3 cache access. Depending on the outcome of this lookup, different operations may have to be invoked on the Pool directory. In the following, we discuss four important operations.

Adding a Sharer

On a sparse directory miss, the replacement process allocates a sparse directory entry for the requested block and encodes the new sharer directly in the pointer P. At this point, the single sharer (S) bit in the sparse directory entry is also set. When a second sharer needs to be added, a pool entry is allocated and the two sharers are encoded using the limited pointer format in the allocated pool entry. The single sharer (S) bit is cleared in the sparse directory entry and the pointer P is updated to point to the allocated pool entry establishing the link between the sparse directory entry's tag and the pool entry. In a general situation, when a new sharer needs to be added to a block B which has already been allocated a number of pool entries, the sharer addition logic explores four possible avenues, as discussed below. Suppose the new sharer falls in segment n of the full-map vector. Let the collection of pool entries already allocated to B be Q. Note that the pool entries in Q are all contiguous. First, the collection Q is looked up to find out if there is an entry already allocated for segment n. If such a pool entry exists, the new sharer is encoded in it. If no such entry exists, the collection Q is looked up to locate a pool entry that is currently using the limited pointer format and has a free pointer. If such a pool entry exists, the new sharer is encoded in the free pointer. When no such free pointer is available, it may be possible to add the new sharer by changing the encoding format of a pool entry. Particularly, when all the sharers encoded in a pool entry using the limited pointer format belong to the same segment of the full-map vector, the encoding of the pool entry is changed to the segment vector format creating new opportunities for finding a pool entry to encode the new sharer. When all these three avenues fail, a new pool entry must be added to the collection Q.

The collection Q can be extended by prepending or appending it with a new pool entry. Let the entry just before the collection Q begins be Q_- and the entry just after the collection Q ends be Q_+. If at least one of Q_- and Q_+ is free, the free entry is added to the collection Q. When both of these entries are occupied, one of them is evicted. The victim is chosen as follows. Suppose each pool entry is K-bit wide. Therefore, in the worst case, a block would need $\lceil C/K \rceil$ entries to encode all its sharers, where C is the number of cores. We divide the pool into equally-sized chunks such that each chunk has $\lceil C/K \rceil$ consecutive entries. If Q_+ and Q_- belong to the same chunk (as Q), we victimize one of them at random. If Q_+ and Q_- belong to two different (adjacent) chunks, we victimize the one that belongs to the chunk which has the larger share of Q. The rationale for this victimization policy is that we let Q grow within the chunk which already has a bigger share of Q so that the interference in the adjacent chunk due to this growth is minimized. When Q_- is used for extending Q, the head (H) bit for the current head entry in Q is cleared and Q_- becomes the new head of the extended collection. The pointer P in the sparse directory entry is also updated to point to Q_-.

In all cases, the sharer addition logic first looks up the sparse directory. On a tag hit, if the single sharer (S) bit in the sparse directory entry is not set, the occupied (O) and the head (H) bits of consecutive $\lceil C/K \rceil$ pool entries are examined starting from the pool entry pointed to by the pointer P. The O and H bit arrays are physically kept separately from the pool array so that they can be quickly examined in parallel using an array of bit manipulation logic blocks. This examination reveals the number of pool entries that must be read out. The maximum number of pool entries that a block can use is $\lceil C/K \rceil$. We provision the pool with two read ports so that the required number of access rounds is bounded by $\lceil C/K \rceil/2$. For the pool configuration modeled in this study, we verify using CACTI [37] that this latency is comfortably hidden under the last-level cache access latency for 22 nm nodes. This latency can be further improved by internally banking the pool and ensuring that not all pool entries of a block are allocated in the same bank. The accessed pool entries are examined and the new sharer is added according to the aforementioned protocol. The longest critical path, encountered infrequently, involves reading of the sparse directory, examining the O and H bit arrays, reading of pool entries, examining the pool entries, extending the collection of pool entries, and updating the new pool entry. For the workloads considered in this study, on average, 98.6% of the allocated sparse directory entries need at most two pool entries and 99% need at most three pool entries. In any case, addition of a new sharer is off the critical path and can be executed after responding to the requesting core.

Removing a Sharer

The coherence protocol modeled by our system generates replacement hints to the directory when a core evicts a shared block. This is done to exclude the already evicted blocks from tracking information so that the information is up-to-date and exact. In addition, dirty evictions always generate writebacks to the L3 cache. Also, the E state evictions necessarily generate a replacement hint to update the tracking information in the directory. In all these cases, the evicting core's identity must be removed from the tracking information. This operation requires looking up the sparse directory followed by pool entry accesses, if needed, and removal of the sharer leading to the possible release of a pool entry. When the number of sharers for a sparse directory entry reduces to one, the sparse directory entry frees its pool entries. The single sharer is encoded directly in the pointer P of the sparse directory entry and the single sharer (S) bit is set. The port requirements are similar to the sharer addition operation. Removal of a sharer is also off the critical path because it can be executed after sending the eviction acknowledgment to the evicting core.

Allocation of the First Pool Entry

A sparse directory entry needs to allocate its first pool entry when the number of sharers of the block that the entry is tracking becomes two. This allocation is treated differently from growing an already allocated collection of pool entries because appropriate positioning of the first allocated pool entry is important so that the future growth of the collection does not lead to pathological conflicts and pool entry evictions. By examining the vector of occupied (O) bits of the pool entries, a free pool entry can be identified. To make the implementation efficient, the occupied bitvector is segmented and one occupied vector is maintained for a segment of $\lceil C/K \rceil$ consecutive pool entries. Each occupied bitvector is also accompanied by a $\lceil \log_2(C/K) \rceil$-bit wide population counter which tracks the number of bits set in the occupied bitvector. The population counter is incremented when an entry in the segment is occupied and decremented when an entry in the segment is freed. Finally, a free entry index is maintained for each segment of pool entries, which points to the next free entry in the segment. By examining the occupied bitvector, the free entry index can be updated. For our configurations, we need at most ten such occupied bitvectors, population counters, and free indices per last-level cache (LLC) bank (same as an L3 cache bank).

To more or less evenly distribute the density of occupied entries over the entire pool, a round-robin policy is utilized for selecting the segment with a free pool entry. An index of the segment from which the last pool entry was allocated is maintained to assist in the selection of a free pool entry. When a free pool entry cannot be obtained, a tail entry is randomly selected for eviction from the round-robin segment. The protocol for evicting a pool entry is discussed next.

Pool Entry Eviction

For evicting a pool entry, the sparse directory entry associated with it needs to be located. Once the sparse directory entry has been located, the sharers encoded in the evicted pool entry are back-invalidated. The occupied (O) and the head (H) bits of the pool entry are cleared. When the sparse directory tag is occupying a single pool entry (the pool entry being evicted), all but one of the sharers encoded in the pool entry are back-invalidated.[2] The single sharer that is not back-invalidated is encoded directly in the pointer P of the sparse directory entry and the single sharer (S) bit in the sparse directory entry is set.

To locate the sparse directory entry associated with the evicted pool entry, the set index stored in the pool entry is used to read out the entire sparse directory set. Once the sparse directory set has been read out, the pointers P stored in all the ways of the sparse directory set are compared against the index of the pool entry. When the head (H) bit is not set for the pool entry being evicted, the index of the head of the collection containing the evicted pool entry is used for comparison. We organize the sparse directory's tag and data arrays as two separate direct-mapped arrays with one row of each array containing the tag and data entries for a set. This allows us to efficiently read out one entire set. Such a design is attractive for a set-associative array if the size of the entire set is small, which is true for Pool directories.

3.3 Simulation Environment

We use a significantly modified version of the Multi2Sim simulator [78] to model a chip-multiprocessor having 128 dynamically scheduled out-of-order issue x86 cores clocked at 2 GHz. Each core has private L1 and L2 caches with the L2 cache being non-inclusive/non-exclusive with respect to the L1 instruction and data caches. The L1 instruction and data caches are 32 KB in size and eight-way set-associative. The unified L2 cache is 128 KB in size and eight-way set-associative. The L1 and L2 cache lookup latencies are one and two cycles, respectively. The cores along with their private caches are arranged on a mesh

[2] We use the term back-invalidation to denote all invalidations received by the private cache hierarchy due to sparse directory entry or pool entry evictions.

interconnect. The L3 cache is shared among all the cores and non-inclusive/non-exclusive with respect to the L1 and L2 caches. Each mesh switch, in addition to having a core along with its L1 and L2 caches, has a bank of the shared L3 cache and a slice of the sparse directory. Each L3 cache bank is 256 KB in size, has sixteen ways, and requires three cycles for lookup. The sparse directory slice in a switch is responsible for tracking the blocks mapped to the L3 cache bank in that switch. The associativity of the directory slice is same as the per-core L2 cache associativity (eight) and the total number of entries in the directory slice is decided relative to the number of L2 cache blocks per core. The ratio of the number of entries in one slice of a $(R)\times$ sparse directory to the number of L2 cache blocks per core is R. All levels of the cache have 64-byte blocks and implement a least-recently-used (LRU) replacement policy. The sparse directory implements a 1-bit not-recently-used (NRU) replacement policy. The simulated system models eight single-channel memory controllers evenly distributed over the mesh. Each memory controller connects to a 2 GB DRAM module modeled using DRAMSim2 [68]. The width of each of the eight channels is 64 bits. Each DRAM module is eight-way banked single-rank DDR3-2133 with 12-12-12 latency parameters and burst length eight. The aggregate DRAM bandwidth is 133.3 GB/s. We found this to be adequate for our workloads running on a 128-core system having a reasonably large shared LLC (32 MB distributed over 128 banks, each bank being 256 KB). The memory controllers implement the FR-FCFS scheduling algorithm.

The applications for this study are drawn from various sources and detailed in Table 3.1 (ROI refers to the parallel region of interest). The inputs, configurations, and simulation lengths are chosen to keep the simulation time within reasonable limits while maintaining fidelity of the simulation results. The PARSEC and SPLASH-2[3] applications are simulated in execution-driven mode, while the rest of the applications are simulated by replaying an instruction trace collected through the PIN tool. The PIN trace is collected on a 24-core machine by running the multi-threaded applications creating at most 128 threads (including client, server, application, and JVM threads). Before replaying

[3] The SPLASH-2 applications are drawn from the SPLASH2X extension of the PARSEC distribution.

Table 3.1: Simulated applications

Suite	Applications	Input/Configuration	Simulation length
PARSEC	dedup, fluidanimate, swaptions	sim-medium, sim-medium, sim-small	Complete ROI
SPLASH-2	barnes	32K particles	Complete ROI
	fft	4M complex doubles	Complete ROI
SPEC JBB	SPEC JBB	82 warehouses, single JVM instance	Six billion instructions
TPC	MySQL TPC-C	10 GB database, 2 GB buffer pool, 100 warehouses, 100 clients	500 transactions
	MySQL TPC-E	10 GB database, 2 GB buffer pool, 100 clients	Five billion instructions
	MySQL TPC-H	2 GB database, 1 GB buffer pool, 100 clients, zero think time, even distribution of Q6, Q8, Q11 Q13, Q16, Q20 across client threads	Five billion instructions
SPEC Web	Apache HTTP server v2.2 running Banking, Ecommerce, Support	Worker thread model, 128 simultaneous sessions, mod_php module	Five billion instructions
SPEC JVM	compiler.sunflow, crypto.aes	Five operations	Five billion instructions in ROI

the trace through the simulated 128-core system, it is pre-processed to expose maximum possible concurrency across the threads while preserving the global order at global synchronization boundaries and between load-store pairs touching the same memory block (64 bytes). Since the tracing is done on a 24-core machine, at a time only 24 threads would be active in the trace. To overcome this limitation and expose the maximum available concurrency in the application, the trace is pre-processed so that the instructions from an arbitrary number of threads can be pushed in parallel into the simulator for replaying at any point in time obeying the instruction fetch bandwidth of each core. This replay mechanism needs to obey all load-store ordering to the same cache block address as observed during tracing to maintain the schedule of replaying the synchronization events, any other inherent races in the application, and true- as well as false-sharing events. Also, it is important to maintain ordering with respect to certain system calls that pertain to thread creation, thread join, etc.. For example, the threads created using a fork/clone call cannot issue any instruction until the creator thread completes the fork/clone call. All these are taken care of in the pre-processing step which essentially marks the ordering points and embeds the corresponding ordering rule at each ordering point in the trace. Additionally, the collected trace already captures the busy-waiting in a spin-lock or flag or barrier synchronization. The amounts of busy-waiting captured in these scenarios correspond to the thread schedule observed during tracing and this is one of the many valid thread schedules. This is the schedule that is replayed through our simulator. We note that some other schedule may have more or less amounts of busy-waiting at the synchronization points.

We evaluate five directory organizations in this study. These are summarized below. We assume a 48-bit physical address.

Scalable coherence directory (SCD) [69]: Sparse directory with $\frac{1}{16}\times$ entries. Each slice has sixteen sets and eight ways. Each directory way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, two limited-pointer fields and their valid bits (total sixteen bits, which can also encode the sharers in a sixteen-core cluster in a hierarchical representation), two bits to encode the type of representation (limited-pointer, root, leaf), and three bits to encode the cluster id in a hierarchical representation. Total size is 128 slices $\times$ 16 sets $\times$ 8 ways $\times$ 55 bits i.e., **110 KB**.

Hybrid directory [27]: Sparse directory with $\frac{1}{16}\times$ entries. Each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), and 1-bit NRU state. Out of the eight ways in a set, two ways can encode full-map sharer vectors and each is of size 128 bits. The remaining six ways can encode a single pointer, each of size seven bits. Total size is **142.5 KB**.

Select directory [85]: Sparse directory with $\frac{1}{16}\times$ entries. Each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, a 7-bit pointer, and a pointer state bit (single sharer vs. pool pointer). Depending on the pointer state bit, the 7-bit pointer field stores either the private owner of a block or the entry id of a dynamically assigned full-map bitvector from a sixteen-entry fully-associative pool of bitvectors. Each pool entry has a 128-bit sharer vector, a valid bit, and four bits of back-pointer to the associated sparse directory set. Total size is 128 slices $\times$ 16 sets $\times$ 8 ways $\times$ 42 bits + 128 slices $\times$ 16 pool entries $\times$ 133 bits i.e., **117.25 KB**. The bitvector pool exercises FIFO replacement.

Pool directory: Sparse directory with $\frac{1}{16}\times$ entries. Each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, a 7-bit pointer, and a pointer state bit (single sharer vs. pool pointer). The pool has 40 entries per slice. Each pool entry has four limited-pointer fields and their valid bits (total 32 bits, which can also encode the sharers in a 32-core cluster), one bit to encode the type of representation (limited-pointer, sharer cluster), one occupied bit, one head bit, two bits to encode the cluster id in a sharer cluster representation, and four bits of back-pointer to the sparse directory set. Total size is 128 slices $\times$ 16 sets $\times$ 8 ways $\times$ 42 bits + 128 slices $\times$ 40 pool entries $\times$ 41 bits i.e., **109.625 KB**. Note that the SCD, Select, and Pool directories are sized to have similar storage overhead.

Full-map directory: Sparse directory with $\frac{1}{16}\times$ entries. Each slice has sixteen sets and eight ways. Each way has a valid bit, a 31-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, and a 128-bit vector. Total size is **324 KB**.

3.4 Simulation Results

In this section, we quantitatively compare the SCD, Hybrid directory, Select directory, Pool directory, and full-map directory in terms of interconnect traffic, volume of private cache misses, the number of sparse directory fills, and overall application performance.

Figure 3.4 shows a comparison of interconnect message count. Each group of bars corresponds to an application and the rightmost group in the bottom panel shows the average. The bars in a group correspond to SCD, Hybrid, Select, Pool, and full-map organizations from left to right. Each bar is divided into three segments representing three different types of messages. The forwarded requests, their responses, invalidations due to writes, and their acknowledgments constitute the coherence messages. The private cache misses, their responses, writebacks, and writeback acknowledgments constitute the processor requests and responses. Back-invalidations induced by sharer evictions from the directory and their acknowledgments constitute the third category of messages. We exclude the messages between the L3 cache banks and the memory controllers from these results because, as expected, the volume of these messages is not affected by changes in the directory organization. For each application, the message counts are normalized with

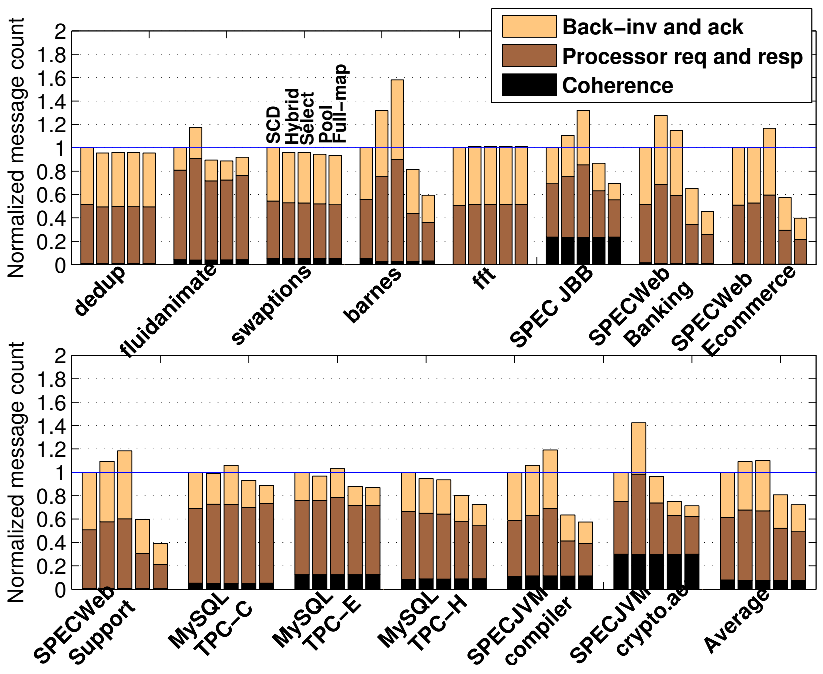

Figure 3.4: Interconnect message count normalized to SCD.

respect to SCD.

Across the board, we observe that the Pool directory organization is able to save a significant volume of interconnect messages. While the coherence message count remains largely constant across the different directory organizations, the primary savings achieved by the Pool directory arise from reduction in private cache misses and back-invalidations. Since the Pool directory manages the directory space more efficiently, the pressure on the directory goes down significantly leading to a less number of back-invalidations. The harmful subset of the back-invalidations causes an increase in the volume of the private cache misses. The savings achieved by the Pool directory are particularly impressive for fluidanimate (11% reduction), barnes (18% reduction), SPEC JBB (13% reduction), SPECWeb (35% to 43% reduction), TPC-E (12% reduction), TPC-H (20% reduction), and the SPEC JVM applications (25% to 36% reduction). In these applications, SCD suffers from high directory pressure because it requires multiple directory entries to encode more than two sharers. This leads to premature invalidation of directory entries tracking active blocks causing an increased volume of private cache misses. On average, the Pool directory organization achieves a 19% reduction in interconnect message count and is able to bridge a big portion of the wide gap between SCD and full-map organizations (see the Average group of bars).

The Hybrid and Select organizations suffer from respectively 9% and 10% average increase in interconnect message count compared to SCD. The Hybrid organization's static partitioning of the directory space among the two types of directory entries fails to match the dynamic demand of directory entries over time and across applications. The Select organization, on the other hand, fails to track all the active shared blocks with the few wide sharer vectors. Figure 3.5 further shows the interconnect traffic (total message size) normalized to SCD. The trends are similar to those shown in Figure 3.4. On average, compared to SCD, the Hybrid and Select organizations experience respectively 6% and 7% more interconnect traffic and the Pool directory organization enjoys a 20% reduction in interconnect traffic.

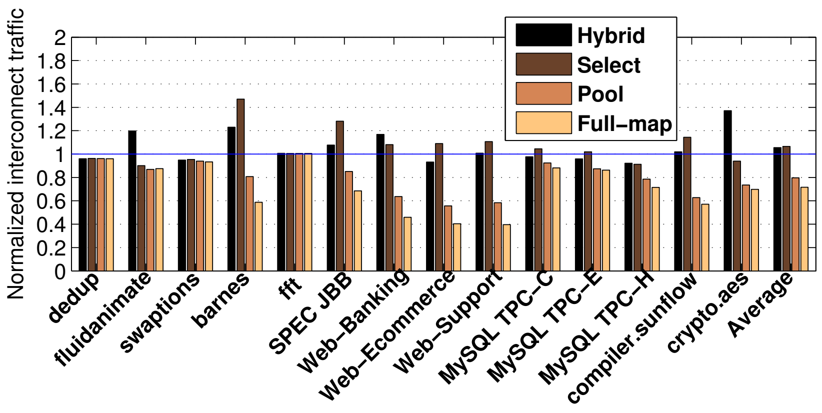

Figure 3.5: Interconnect traffic normalized to SCD.

To further understand the directory pressure in SCD, Figure 3.6 quantifies the number of sparse directory entry allocations (not to be confused with pool entry allocations in the Pool and Select directory organizations) normalized to SCD. Across the board, we see that the SCD organization experiences a high volume of directory allocations indicating a significant amount of directory conflicts. The rest of the organizations all have almost equal number of sparse directory entry allocations, as expected. Only SCD requires multiple directory entries for encoding the sharers of a single block in a hierarchical manner.

On average, SCD suffers from almost double the number of directory entry allocations compared to the other four organizations. The applications that enjoy significant savings in message count and traffic with Pool directory are also the ones that experience relatively higher volume of sparse directory allocations in SCD (e.g., fluidanimate, barnes, SPEC JBB, SPECWeb, TPC-E, TPC-H, and the SPEC JVM applications). Even though the Hybrid, Select, Pool, and full-map organizations have nearly the same number of tag allocations in the sparse directory, their differences in the message traffic arise due to the eviction of sharer tracking information from the tags and not due to eviction of tags from the sparse directory (e.g., eviction of a pool entry in the Pool and Select directories or a swap between bitvector and pointer ways in the Hybrid directory).

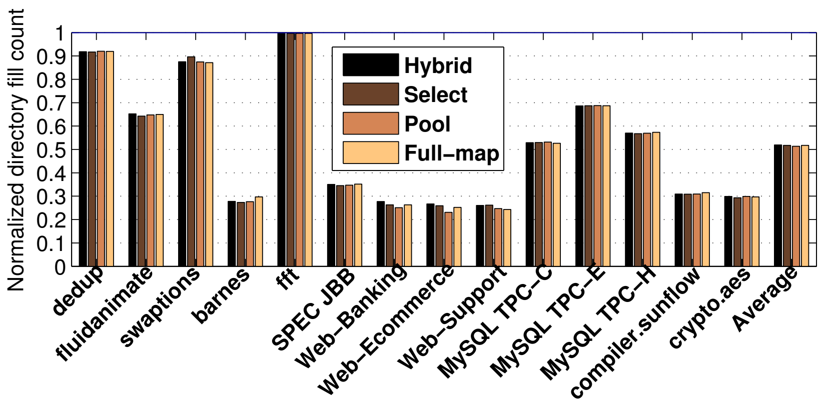

Figure 3.6: Sparse directory fill count normalized to SCD.

Figure 3.7 presents the volume of private cache misses normalized to SCD. We also show the breakdown of private cache misses into code and data misses. These results closely correlate with the processor request and response message count data shown in Figure 3.4. The Pool directory organization, on average, enjoys 19% less private cache misses compared to SCD, while the Hybrid and Select organizations suffer from respectively 9% and 10% increase in the volume of private cache misses. The Pool directory is able to save both code and data misses, while the Hybrid and Select organizations suffer primarily due to increased volumes of code misses compared to SCD. Since the code blocks experience good amount of sharing, these results indicate that the Hybrid and Select organizations are unable to track all the actively shared code blocks with their resources for tracking

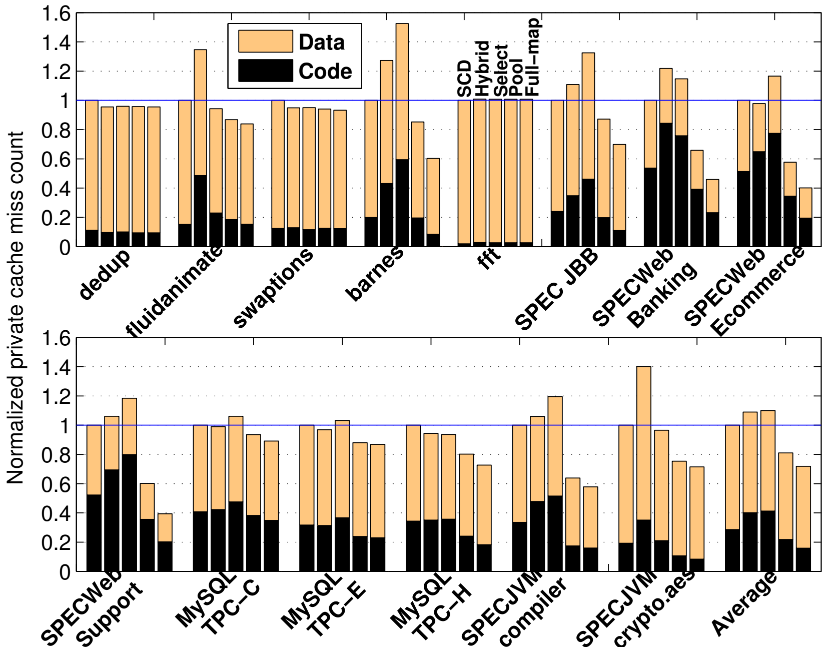

Figure 3.7: Private cache miss count normalized to SCD.

shared blocks.

Figure 3.8 summarizes the performance speedup achieved by the Hybrid, Select, Pool, and full-map directory organizations over SCD. The performance improvement achieved by the Pool directory organization is 5%, on average; significant gainers are fluidanimate (8%), barnes (16%), SPECWeb (5% to 7%), TPC-H (5%), and SPEC JVM compiler.sunflow (20%). Most importantly, while using only one-third of the directory space of a full-map organization, the average performance of the Pool directory organization comes within 2.4% of the full-map directory organization. The Hybrid and Select directory organizations, on average, perform respectively 2% and 4% worse than the SCD organization.

For a budget-constrained sparse directory such as $\frac{1}{16}\times$, the directory organization and the directory replacement policy may play an important role in determining the end-performance. In the following, we evaluate a sparse directory design that uses a four-way skew-associative organization with the timestamp-based three-level least-recently-used Z-cache replacement protocol (52 replacement candidates) [69]. Table 3.2 summarizes the in-

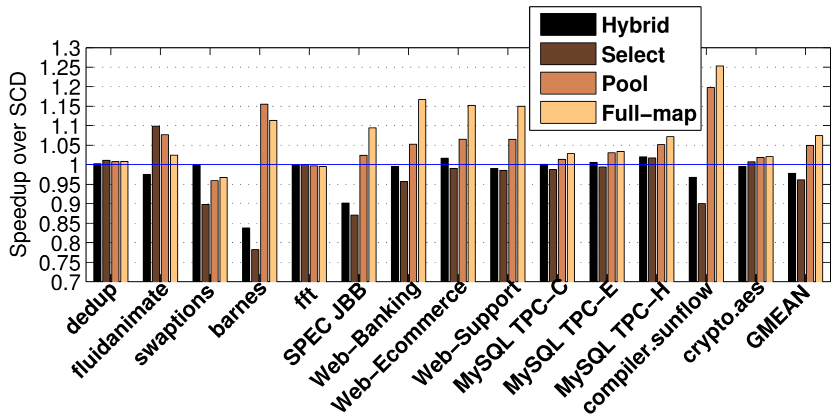

Figure 3.8: Speedup over SCD.

terconnect traffic, speedup, and dynamic energy expended by the directory structures (using 22 nm nodes) for the Pool directory design. The results are averaged over all the applications and normalized to SCD. The results are shown for two different organizations of the $\frac{1}{16}\times$ sparse directory, namely, eight-way set-associative exercising NRU replacement (this is the design we have been discussing so far) and four-way Z-cache. While the Z-cache organization slightly reduces the performance gap between SCD and Pool directory, the latter continues to save 16% interconnect traffic, on average. The Pool directory saves 85% dynamic energy in the coherence directory reads and writes compared to SCD in the set-associative organization. There are two primary reasons for this large saving. First, the Pool directory-based design enjoys 19% less private cache misses leading to a significantly reduced volume of directory accesses. Second, due to the hierarchical encoding, the SCD design may require multiple set-associative lookups into the sparse directory array to complete one private cache miss request. Although the Pool directory may also need multiple lookups into the pool, the expended energy is significantly less due to the tagless direct-mapped design of the pool. For the Z-cache organization, the Pool directory continues to save 82% energy compared to SCD, on average.

3.4.1 Sensitivity to Directory Storage Budget

In this section, we examine the impact of increasing the sparse directory size from $\frac{1}{16}\times$ to $\frac{1}{8}\times$. Since the Hybrid and Select directory organizations perform close to each other, we

Table 3.2: Pool directory results relative to SCD for $\frac{1}{16}\times$ directory

Organization	Traffic	Speedup	Dynamic energy
Set-assoc., NRU	0.80	1.05	0.15
Z-cache	0.84	1.04	0.18

consider only one of them, namely the Hybrid organization, for the following studies. We quantitatively compare our Pool directory proposal with the SCD, Hybrid, and full-map organizations. The storage investment for each organization is summarized below. We assume a 48-bit physical address.

Scalable coherence directory (SCD) [69]: Sparse directory with $\frac{1}{8}\times$ entries. Each slice has 32 sets and eight ways. Each directory way has a valid bit, a 30-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, two limited-pointer fields and their valid bits (total sixteen bits, which can also encode the sharers in a sixteen-core cluster in a hierarchical representation), two bits to encode the type of representation (limited-pointer, root, leaf), and three bits to encode the cluster id in a hierarchical representation. Total size is 128 slices $\times$ 32 sets $\times$ 8 ways $\times$ 54 bits i.e., **216 KB**.

Hybrid directory [27]: Sparse directory with $\frac{1}{8}\times$ entries. Each slice has 32 sets and eight ways. Each way has a valid bit, a 30-bit tag, 1-bit coherence state (M/E vs. shared), and 1-bit NRU state. Out of the eight ways in a set, two ways can encode full-map sharer vectors and each is of size 128 bits. The remaining six ways can encode a single pointer, each of size seven bits. Total size is **281 KB**.

Pool directory: Sparse directory with $\frac{1}{8}\times$ entries. Each slice has 32 sets and eight ways. Each way has a valid bit, a 30-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, a 7-bit pointer, and a pointer state bit (single sharer vs. pool pointer). The pool has 76 entries per slice. Each pool entry has four limited-pointer fields and their valid bits (total 32 bits, which can also encode the sharers in a 32-core cluster), one bit to encode the type of representation (limited-pointer, sharer cluster), one occupied bit, one head bit, two bits to encode the cluster id in a sharer cluster representation, and five bits of back-pointer to the sparse directory set. Total size is 128 slices $\times$ 32 sets $\times$ 8 ways $\times$

41 bits + 128 slices × 76 pool entries × 42 bits i.e., **213.875 KB**. Note that the SCD and Pool directories are sized to have similar storage overhead.

Full-map directory: Sparse directory with $\frac{1}{8}\times$ entries. Each slice has 32 sets and eight ways. Each way has a valid bit, a 30-bit tag, 1-bit coherence state (M/E vs. shared), 1-bit NRU state, and a 128-bit vector. Total size is **644 KB**.

Figure 3.9 shows a comparison of interconnect message count. For each application, the message counts are normalized with respect to SCD. Across the board, the Pool directory organization continues to save a significant volume of interconnect messages. On average, the Pool directory organization achieves a 15% reduction in interconnect message count. For reference, this saving was 19% for a $\frac{1}{16}\times$ directory. The Hybrid organization suffers from 10% average increase in interconnect message count compared to SCD. Figure 3.10 further shows the interconnect traffic (total message size) normalized to SCD. The trends are similar to those shown in Figure 3.9. On average, compared to SCD, the Hybrid organization experiences 8% more interconnect traffic and the Pool directory organization enjoys a 16% reduction in interconnect traffic. This saving was 20% for $\frac{1}{16}\times$ directory.

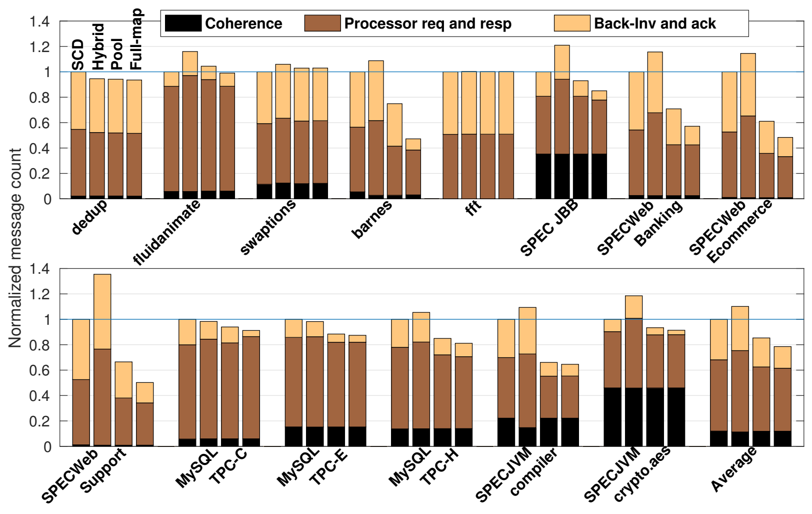

Figure 3.9: Interconnect message count normalized to SCD for $\frac{1}{8}\times$ directory.

Figure 3.11 quantifies the number of sparse directory entry allocations normalized to SCD. Even with increased storage budget, the SCD organization continues to experience a high volume of directory allocations indicating a significant amount of directory conflicts.

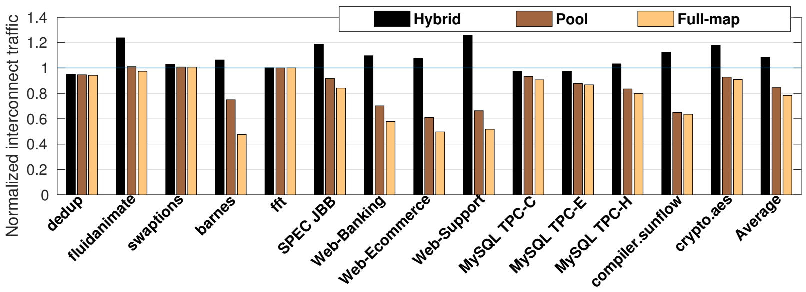

Figure 3.10: Interconnect traffic normalized to SCD for $\frac{1}{8}\times$ directory.

The rest of the organizations all have almost equal number of sparse directory entry allocations, as expected.

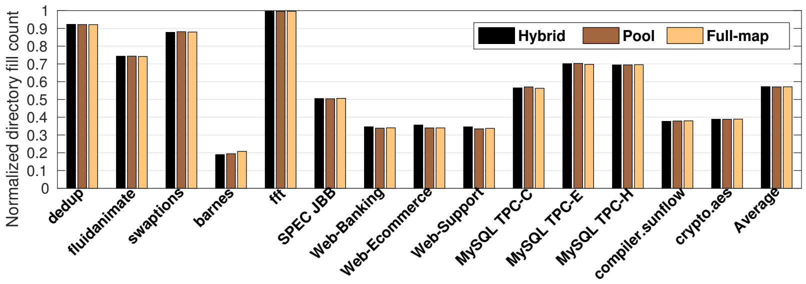

Figure 3.11: Sparse directory fill count normalized to SCD for $\frac{1}{8}\times$ directory.

Figure 3.12 presents the volume of private cache misses normalized to SCD. We also show the breakdown of private cache misses into code and data misses. These results closely correlate with the processor request and response message count data shown in Figure 3.9. The Pool directory organization, on average, enjoys 14% less private cache misses compared to SCD (this saving was 20% for a $\frac{1}{16}\times$ directory), while the Hybrid organization suffers from a 12% increase in the volume of private cache misses.

Figure 3.13 summarizes the performance speedup achieved by the Hybrid, Pool, and full-map directory organizations over SCD. The performance improvement achieved by the Pool directory organization is 3%, on average (this speedup was 5% for a $\frac{1}{16}\times$ directory); significant gainers are SPECWeb (4% to 5%) and SPEC JVM compiler.sunflow (13%). Most importantly, while using about one-third of the directory space of a full-map organization, the average performance of the Pool directory organization comes within 2.1% of the full-map directory organization. The Hybrid directory organization, on average, performs 1% worse than the SCD organization.

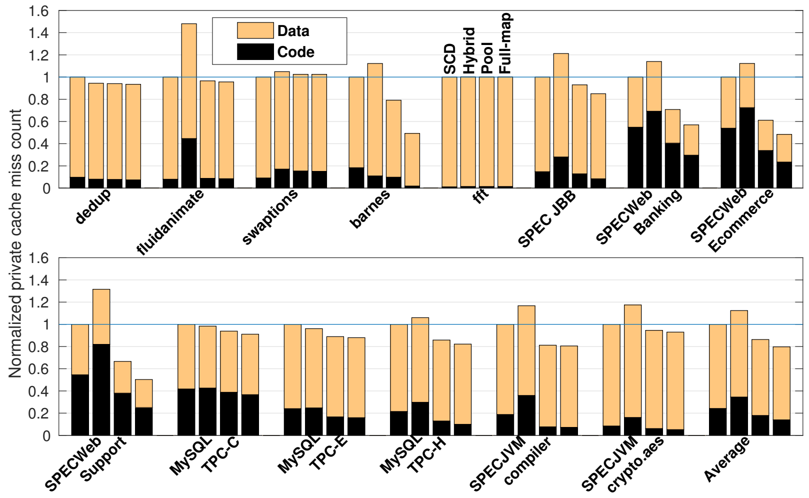

Figure 3.12: Private cache miss count normalized to SCD for $\frac{1}{8}\times$ directory.

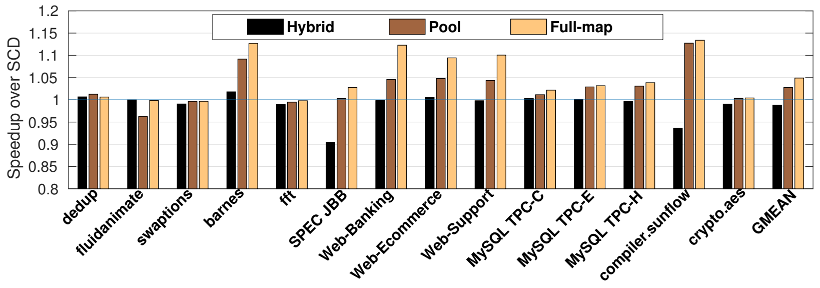

Figure 3.13: Speedup over SCD for $\frac{1}{8}\times$ directory.

3.5 Summary

We have presented a novel coherence directory organization that has a set-associative sparse directory and a direct-mapped pool, each entry of which can act as a limited-pointer entry as well as a short sharer vector entry encoding the sharers in a cluster of cores. A dynamically allocated collection of such pool entries can efficiently track all the sharers of a block. Each sparse directory entry has a pointer, which can either encode a sharer (useful for tracking private blocks) or point to a pool entry. The pool entries allocated to a shared block are contiguously placed in the pool so that maintaining a pointer to the head entry is enough. Simulation results on a 128-core chip-multiprocessor show that our proposal performs 5% better than the state-of-the-art dynamic hierarchical directory organization while reducing the interconnect traffic by 20%. Our proposal delivers performance within

2.4% of a full-map organization while consuming only one-third of the directory space of a full-map organization.

Chapter 4

Tiny Directory

The number of sparse directory entries is an important determinant of end-performance. An undersized sparse directory may experience premature eviction of tracking entries leading to back-invalidation of the blocks corresponding to the evicted tracking entries. At the same time, the large number of entries in an over-provisioned sparse directory can present a significant hurdle to scalability. In this chapter, we explore the problem of optimizing the number of sparse directory entries while maintaining performance. We begin our discussion with a motivating study that quantifies the importance of the number of entries in the sparse directory of a 128-core system. This study clearly brings out the challenge involved in bringing down the number of sparse directory entries.

Figure 4.1 shows the execution time of seventeen multi-threaded applications as the number of entries in the sparse directory is varied in a 128-core system. The results are normalized to the execution time with a $2\times$ sparse directory. All sparse directories are eight-way set-associative. [1] We would like to remind the readers that an $(R)\times$ sparse directory has R times as many entries as the aggregate number of L2 cache blocks across all cores. In these experiments also, we use a three level cache hierarchy with L1 and L2 caches being private to the cores and L3 cache being shared by all cores. The L2 cache is non-inclusive/non-exclusive of the L1 cache and the L3 cache is non-inclusive/non-exclusive of both L1 and L2 caches. On average, the execution time with $\frac{1}{4}\times$, $\frac{1}{8}\times$, and $\frac{1}{16}\times$ sparse

[1] Section 4.2 discusses our simulation environment.

directories increases by 3%, 11% and 28%, respectively, compared to the 2× directory. Ocean_cp is an outlier and improves in performance with decreasing directory size because a smaller directory converts a subset of performance-critical three-hop accesses to two-hop accesses. Overall, reducing the coherence tracking overhead without losing performance is challenging, yet important.

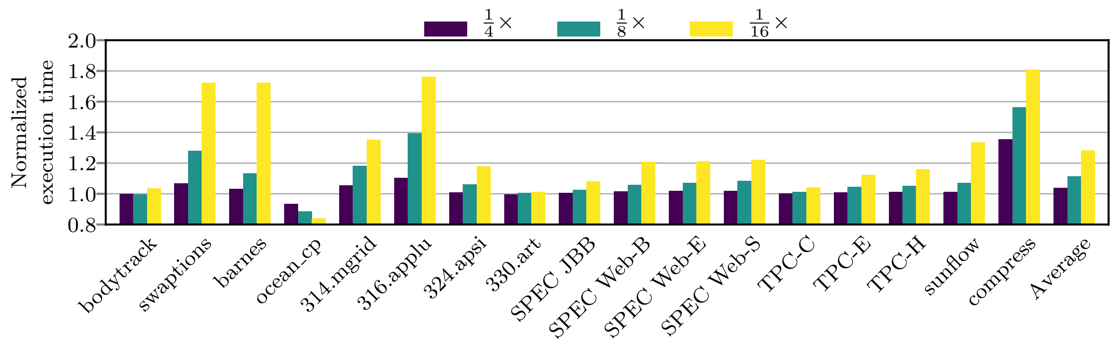

Figure 4.1: Performance with $\frac{1}{4}\times$, $\frac{1}{8}\times$, and $\frac{1}{16}\times$ sparse directories normalized to a 2× directory.

A block allocated in the LLC (same as the L3 cache) either remains private throughout its residency in the LLC or gets actively shared. To understand the proportion of these two types of blocks, Figure 4.2 shows the percentage of the allocated LLC blocks that experience a maximum of k distinct sharers during the residency in the LLC where k falls in four possible sharer count bins: 2 to 4, 5 to 8, 9 to 16, and 17 to 128 (end-points inclusive). These data are collected on a 128-core system. The LLC is sized so that the number of blocks is same as the number of entries that a 2× sparse directory would have. These data show that, on average, 21% of the allocated blocks observe sharing, while the rest remain private during their residency in the LLC.[2] We note that a block that is privately cached by core X and later privately cached by core X' is recorded as a

[2] The left panel of Figure 3.1 in Chapter 3 showed a completely different piece of data and the readers should not attempt to draw any correspondence between that and Figure 4.2. The left panel of Figure 3.1 showed the distribution of the allocated *sparse directory entries* based on sharer count. For example, it showed that the shared blocks of swaptions and barnes respectively occupy 37% and 25% of all allocated entries of a 2× sparse directory. On the other hand, Figure 4.2 shows that these two applications have 36% and 89% of shared blocks among all blocks allocated in the LLC. It is impossible to compare these two pieces of data and draw any useful conclusion because an LLC block can undergo multiple episodes of sharing while in the LLC. Each such episode would allocate a sparse directory entry at different points in time. Therefore, the total number of sparse directory allocations is at least as large as the total number of LLC allocations. As a result, for an application, the percentage shown in the left panel of Figure 3.1 in Chapter 3 can be more or less compared to the percentage shown in Figure 4.2.

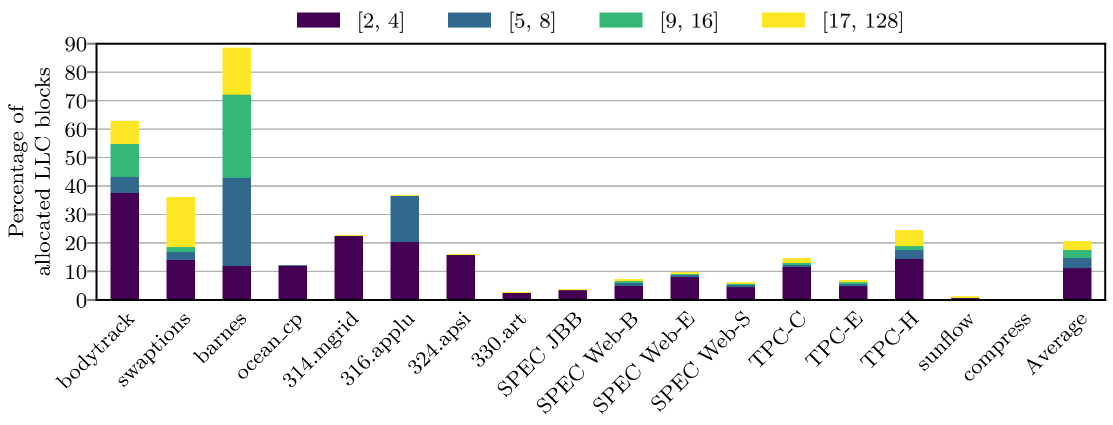

Figure 4.2: Distribution of maximum sharer count per allocated LLC block.

private block in the data shown in Figure 4.2. In the literature, such blocks are referred to as temporally private [5, 86]. Such blocks require tracking of just one core at a time in the sparse directory entry. While these data do not show the absolute shared footprint, the SPECWeb and TPC benchmark applications have much larger shared footprints than most of the applications and they also carry out a larger number of LLC fills.

The primary observation from Figure 4.2 is that a vast majority of the blocks allocated in the LLC remain either private or temporally private during their residency in the LLC. Motivated by this observation, recent proposals have explored several ways to reduce the number of sparse directory entries that track private blocks [5, 24, 26, 27, 86]. Some of these techniques include the following: tracking coherence of private regions at a coarse grain and switching to multi-grain tracking when a block in such a region gets shared [5, 10, 27, 86]; not tracking coherence of private pages identified by the operating system and falling back to block-grain tracking through an expensive recovery mechanism when the first sharer of such a page is detected [24]; speculating that the private entries evicted from the sparse directory will remain private and not tracking them until they get shared when an expensive broadcast-based recovery mechanism reconstructs the sharer information [26].

The data in Figure 4.2 also indicate that if a sparse directory is dedicated to track only shared blocks, it can be small. We conduct an experiment to find out how small such a sparse directory can be. In this experiment, a block's tracking entry is allocated in the sparse directory only when the block enters the shared state with two distinct sharers.

The tracking entry stays in the sparse directory until it is evicted from the directory or the block reaches a state where it has no sharer or owner.[3] As long as a block remains private/temporally private or is exclusively owned by a core, it is tracked in a special structure of unbounded capacity. It is important to note that if a block exhibits a sharing pattern where it moves from one core to another while staying in an exclusively owned state (E or M in our baseline MESI protocol), it is tracked in the special unbounded structure and does not get allocated in the sparse directory until and unless it enters the S state with two sharers.

Figure 4.3 quantifies the performance of such a design with varying size of the sparse directory dedicated to track only shared blocks. These results completely ignore the overhead of the unbounded special structure that tracks the other blocks. As the size of the sparse directory dedicated to track only shared blocks is set to $\frac{1}{16}\times$, $\frac{1}{32}\times$, $\frac{1}{64}\times$, and $\frac{1}{128}\times$, the average losses in performance compared to a traditional $2\times$ sparse directory are 1%, 4%, 13%, and 28%, respectively. The $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sparse directories are eight-way set-associative, while the $\frac{1}{128}\times$ sparse directory having just sixteen entries per LLC bank is organized such that each slice is a fully-associative sixteen-entry cache. We have also conducted this experiment with a four-way skew-associative sparse directory that employs an H3 hash-based Z-cache organization [70] for the $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sizes. The $\frac{1}{16}\times$ sparse directory uses 52 replacement candidates, while $\frac{1}{32}\times$ and $\frac{1}{64}\times$ sparse directories use 16 replacement candidates. We use global LRU replacement policy for these skew-associative directories. With skew-associative organization, the performance losses are 0.5%, 3%, and 12% respectively for $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sizes of the sparse directory. These results indicate that even if the entire tracking overhead of non-shared blocks is lifted from the sparse directory, it is not possible to reduce the directory size to $\frac{1}{32}\times$ or less using traditional techniques without suffering from noticeable performance losses. In particular, achieving a sparse directory overhead in the range of $\frac{1}{32}\times$ to $\frac{1}{256}\times$, which is the target of this work, appears quite challenging. In the following, we summarize our approach to this problem.

[3] In our implementation, all evictions from the private cache hierarchy are notified to the directory [62]. The eviction notices for the blocks in E or S state do not carry any data, as already discussed in Chapter 2.

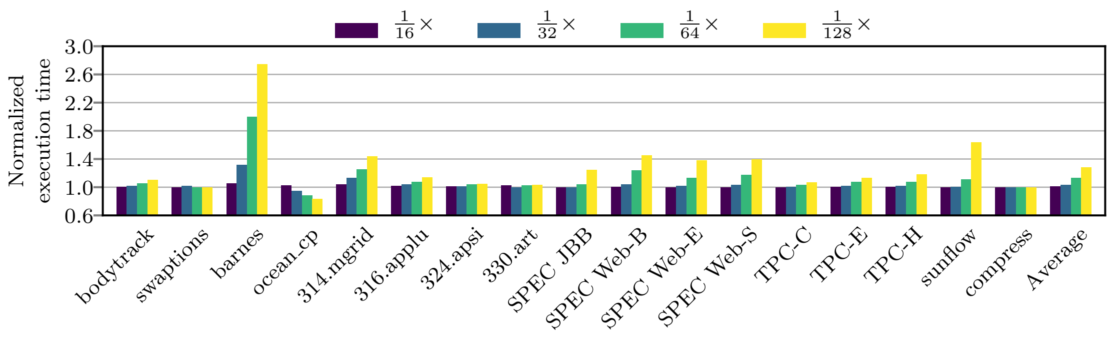

Figure 4.3: Performance with $\frac{1}{16}\times$, $\frac{1}{32}\times$, $\frac{1}{64}\times$, and $\frac{1}{128}\times$ sparse directories for tracking shared blocks only. Tracking non-shared blocks has no overhead. Results are normalized to a $2\times$ sparse directory.

In this chapter, we present a novel ground-up approach for designing a robust sparse directory having a minimal number of entries. We retain the simplicity and scalability of a traditional broadcast-free OS-independent block-grain coherence protocol. We begin our exploration with an architecture that does not have a sparse directory and consider the possibility of borrowing a few bits of the LLC data way of the block for tracking coherence information (Section 4.3). For private blocks, this in-LLC coherence tracking works quite well and successfully rids the sparse directory of the overhead of tracking private blocks. In this design, a significant performance problem arises when a block gets shared. Each sharing access received by the LLC must be forwarded to an elected sharer, which can supply the data block to the requester; the LLC cannot supply the correct data block because portion of the LLC data block is corrupted and used to track the sharers. Due to this problem, the critical path of a large volume of shared accesses can get extended to three transactions (traditionally referred to as three hops) from two transactions (or two hops). We address this performance shortcoming by architecting a tiny directory, which is a novel sparse directory design for dynamically identifying and tracking a critical subset of the blocks that experience most shared accesses (Section 4.4). We also propose a few different policies for managing the contents of the tiny directory. Since the optimal size of the tiny directory depends on the dynamic working set size of the shared blocks in an application, it is not possible for a designer to know this size beforehand. To make the tiny directory proposal address this problem in a robust fashion, we introduce the option of selectively spilling a subset of the shared tracking entries into the LLC space when the tiny directory is too small to track the critical shared working subset. This option, however, introduces the new challenge of dynamically deciding the appropriate volume of spills so that the volume of LLC misses does not get affected. We address this challenge by carefully deciding the subset of the tracking entries that is eligible for spilling into the LLC so that the critical path of core requests does not get lengthened. Simulation results show that our proposal implemented in a 128-core system operating with a tiny directory of size ranging from $\frac{1}{32}\times$ to $\frac{1}{256}\times$ performs within a percentage of a system with a traditional $2\times$ sparse directory (Section 4.5).

4.1 Related Work

The early proposals focused on optimizing the coherence directory store in the distributed shared memory multiprocessor architectures. The first proposal on coherence directory design introduced a bitvector as the directory element [14]. Since then several designs have been proposed to optimize the coherence directory storage in the distributed shared memory multiprocessors [1, 2, 4, 15, 16, 19, 21, 34, 35, 42, 61, 64, 65, 80].

More recent proposals have focused on directory space optimization for CMPs. Several proposals have attempted to optimize the number of entries in the sparse directory. Smart hash functions and skew-associative organizations for the sparse directory have been proposed [29, 69]. Designs that store the evicted directory entries in a memory-resident hash table and delay invalidations have also been explored [49]. Page-grain classification between private and shared data has been used to exclude private blocks from coherence tracking, thereby effectively increasing the number of available directory entries for tracking shared data [24]. A recently proposed design does not invalidate private blocks on directory eviction, but resorts to broadcast when such a block gets shared after the tracking entry of the block is evicted from the sparse directory [26]. Recent proposals employing coarse-grain coherence tracking for privately cached regions can further reduce the required number of directory entries [5, 10, 27, 86]. Proposals that track a small set of sharing patterns and link each active directory entry to a sharing pattern have been explored [89, 90]. The recently proposed in-cache coherence tracking design uses the entire LLC data block of an LLC tag for tracking coherence information for that tag [31]. As we show in Section 4.3, a design similar to this proposal suffers from a large volume of three-hop transactions for shared accesses. Compiler-generated hints about private data have been used to optimize directory allocation [57]. Data-race-free software, disciplined parallel programming models, and self-invalidation of shared data at synchronization boundaries have been used to significantly reduce the coherence directory size or completely eliminate the coherence directory [20, 67, 76].

In this study, we assume each sparse directory entry to be a full-map bitvector and focus squarely on optimizing the number of entries in the sparse directory. However, there have been several proposals that optimize the average number of bits per directory entry including our Pool Directory proposal [27, 55, 69, 71, 85, 87, 88]. Any standard technique for limiting the width of the directory entry can be seamlessly applied on top of our proposal to further reduce the area of the sparse directory.

4.2 Simulation Framework

We use an in-house modified version of the Multi2Sim simulator [78] to model a chip-multiprocessor having 128 dynamically scheduled out-of-order issue x86 cores clocked at 2 GHz. The details are presented in Table 4.1. We explore an array of sparse directory slice configurations. We consider eight-way associative sparse directory slices for all configurations except when a slice contains only eight and sixteen directory entries ($\frac{1}{256}\times$ and $\frac{1}{128}\times$, respectively). In these two cases, we configure each sparse directory slice to be fully-associative. The interconnect switch microarchitecture assumes a four-stage routing pipeline with one cycle per stage at 2 GHz clock. The stages are routing computation, virtual channel allocation, output port allocation, and traversal through switch crossbar. There is an additional 1 ns link latency to copy a flit from one switch to the next. The overall hop latency is 3 ns. The applications for this study are drawn from various sources and detailed in Table 4.2 (ROI refers to the parallel region of interest). The inputs, configurations, and simulation lengths are chosen to keep the simulation time within reasonable limits while maintaining fidelity of the simulation results. The PARSEC, SPLASH-2, and OMP applications are simulated in execution-driven mode, while the rest of the applications are simulated by replaying an instruction trace collected through the PIN tool capturing all activities taking place in the application address space. The PIN trace is collected on a 24-core machine by running each multi-threaded application creating at most 128 threads (including server, application, and JVM threads). Before replaying the trace through the simulated 128-core system, it is pre-processed to expose maximum possible concurrency across the threads while preserving the global order at global synchronization boundaries and between load-store pairs touching the same memory block (64 bytes).

Table 4.1: Simulation environment

On-die cache hierarchy, interconnect, and coherence directory
Per-core iL1 and dL1 caches: 32 KB, 8-way, 2 cycles
Per-core unified L2 cache: 128 KB, 8-way, 3 cycles, non-inclusive/non-exclusive, fill on miss, no back-inval. on eviction
Shared L3 cache: 32 MB, 16-way, 128 banks, bank lookup latency 4 cycles for tag + 2 cycles for data, non-inclusive/non-exclusive, fill on miss, no back-inval. on eviction
Cache block size, replacement policy at all levels: 64 bytes, LRU
Interconnect: 2D mesh clocked at 2 GHz, two-cycle link latency (1 ns), four-cycle pipelined routing per switch (2 ns latency); Each hop: a core, its L1 and L2 caches, one L3 cache bank, one sparse directory slice tracking home blocks.
Sparse directory slice: 1-bit NRU replacement, 8-way (fully-associative for $\frac{1}{128}\times$ and $\frac{1}{256}\times$ sizes)
Coherence protocol: write-invalidate MESI
Main memory
Memory controllers: eight single-channel DDR3-2133 controllers, evenly distributed over the mesh, FR-FCFS scheduler
DRAM modules: modeled using DRAMSim2 [68], 12-12-12, BL=8, 64-bit channels, one rank/channel, 8 banks/rank, 1 KB row/bank/device, x8 devices, open-page policy

Table 4.2: Simulated applications

Suite	Applications	Input/Configuration	Simulation length
PARSEC	bodytrack	sim-medium	Complete ROI
	swaptions	sim-small	
SPLASH-2[4]	barnes	32K particles	Complete ROI
	ocean_cp	514×514 grid	
SPEC OMPM2001	314.mgrid	ref input	One charge, one iteration
	316.applu	train input	Six pseudo-time-steps
	324.apsi	train input	One time-step
	330.art	train 2 inputs	Complete parallel section
SPEC JBB	SPEC JBB	82 warehouses, single JVM instance	Six billion instructions
TPC	MySQL TPC-C	10 GB database, 2 GB buffer pool, 100 warehouses, 100 clients	500 transactions
	MySQL TPC-E	10 GB database, 2 GB buffer pool, 100 clients	Five billion instructions
	MySQL TPC-H	2 GB database, 1 GB buffer pool, 100 clients, zero think time, even distribution of Q6, Q8, Q11 Q13, Q16, Q20 across client threads	Five billion instructions
SPEC Web	Apache HTTP server v2.2 running Banking, Ecommerce, Support	Worker thread model, 128 simultaneous sessions, mod_php module	Five billion instructions
SPEC JVM	compiler.sunflow, compress	Five operations	Five billion instructions in ROI

[4] The SPLASH-2 applications are drawn from the SPLASH2X extension of the PARSEC distribution.

4.3 In-LLC Coherence Tracking

This section discusses the design of an in-LLC coherence tracking technique which does not have a sparse directory and borrows few bits of the LLC block for tracking coherence information. The design extends a traditional MESI coherence protocol [56]. Section 4.3.1 discusses the organization of the coherence states in the LLC. We carefully construct the state encoding so that no new LLC or private cache state bits are required to be introduced. Even a single bit per LLC block or private cache block can lead to a very large storage overhead in the large-scale systems we are dealing with. Section 4.3.2 introduces the small extensions needed on top of the traditional MESI coherence protocol. We evaluate the in-LLC coherence tracking technique in Section 4.3.3 and understand the major shortcomings of this design. This evaluation sets the stage for the tiny directory design, which is the central contribution of this chapter.

4.3.1 Organization of Coherence States

A valid LLC block can be in one of three stable coherence states: unowned/non-shared, exclusively owned by a core (in E or M state), and shared by one or more cores. Additionally, a pending/busy state is needed to handle transience. As in the baseline, we assume two state bits per LLC block: valid (V) and dirty (D). These two bits are used to encode four states of an LLC block as depicted in Table 4.3. The state encoding shown in the last row is introduced for the purpose of in-LLC coherence tracking. In this state, the first four bits (denoted b_0, b_1, b_2, b_3) of the data block encode the extended state of the block as shown in Table 4.4. The number of cores is assumed to be C. In summary, when the LLC block state is (V=0, D=1), either $4 + \lceil \log_2(C) \rceil$ bits or $4 + C$ bits of the data block are corrupted for tracking the extended coherence states.

Table 4.3: LLC block states

V	D	State
0	0	Invalid
1	0	Valid, not modified, unowned, not shared
1	1	Valid, modified, unowned, not shared
0	1	Valid, either owned by a core or shared, part of data block used for extended state encoding

Table 4.4: LLC block extended states

Bit	State
b_0	Dirty
b_1	Pending/Busy
b_2	Exclusively owned ($b_2 = 1$) or shared ($b_2 = 0$)
b_3	Sharer encoding format: If $b_3 = 1$ then bits $b_4, \ldots, b_{3+\lceil \log_2(C) \rceil}$ encode a sharer/owner. If $b_3 = 0$ then bits $b_4, \ldots, b_{3+C}$ encode a C-bit sharer bitvector.

4.3.2 Coherence Protocol Extensions

The in-LLC coherence tracking mechanism minimally extends a traditional baseline write-invalidate MESI coherence protocol. In the baseline protocol, an instruction read access to the LLC is always responded to in S state even if the requester is the only core accessing the block. This helps accelerate code sharing.[5] The baseline protocol assumes that all evictions from the private cache hierarchy are notified to the LLC [62]; the eviction notices for clean blocks do not carry any data. A request that is forwarded to an owner core is responded directly to the requester core with a notification to the home LLC bank for clearing the busy/pending state of the involved cache block. As in the AlphaServer GS320 protocol, a late intervention in the baseline protocol is resolved by the owner core by keeping the evicted block in a buffer until the eviction notice is acknowledged by the home LLC bank [33]. These protocol actions have been discussed in detail in Chapter 2.

In the in-LLC coherence tracking mechanism, an invalid (V=0, D=0) or unowned (V=1) LLC block enters a corrupted state (V=0, D=1) when it is requested by a core. If the access is an instruction read access, the block transitions to the corrupted shared state ($b_2 = 0$); otherwise it transitions to the corrupted exclusive state ($b_2 = 1$). The core id of the requester is recorded using the pointer format ($b_3 = 1$).

A read access to a block in the corrupted exclusive state further changes the state of the block to the corrupted shared state, and the requester obtains the data block from the exclusive owner. A read access to a block in the corrupted shared state leaves the block in the same state, and one of the sharers is elected on-the-fly to supply the data block to the requester. In this case, the critical path of the access increases to three hops (requester to home LLC bank, home LLC bank to the elected sharer, and elected sharer to the requester), instead of two hops in the baseline protocol (LLC would have supplied the data block in the baseline). The sharers are recorded using the bitvector format ($b_3 = 0$).

A read-exclusive access to a block in the corrupted exclusive or corrupted shared state is handled similarly. In the latter case, in addition to electing a sharer to supply the

[5] Existing code blocks may get written to during JIT compilation, dynamic linking, and self-modification of code. These accesses come to the LLC as *data* writes and are handled as usual like normal data writes.

data block, all sharers are invalidated and the LLC block is switched to the corrupted exclusive state. The invalidation acknowledgements are collected at the requester. In this case, the critical path does not increase because even in the baseline, the invalidation acknowledgements from the sharers must be collected at the requester before the request can complete.[6] In the in-LLC protocol, one of these invalidation acknowledgements is of a special type and carries the required data block. An upgrade access to a block in the corrupted shared state invalidates the sharers and the LLC block transitions to the corrupted exclusive state.

An E state eviction (common case for clean private blocks) notification from the private cache hierarchy carries the first $4 + \lceil \log_2(C) \rceil$ bits of the evicted data block to the LLC so that the LLC can reconstruct the block. An M state eviction notification from the private cache hierarchy carries the full data block to the LLC, as usual. An S state eviction notification from the private cache hierarchy does not carry any data with it, as in the baseline. In all cases, the evicting core holds the block in a buffer until it receives an acknowledgement from the LLC. This is required to resolve late intervention races. On receiving an eviction notice from the last sharer of a block in S state, the LLC sends a special eviction acknowledgement to the sharer requesting it to send the portion of the block necessary for reconstruction. The sharer supplies the requested portion from the buffer where the block is held and then de-allocates the buffer entry unless the eviction acknowledgement message has indicated that there is an in-flight forwarded request for the block that must be responded to from the buffer (please refer to Chapter 2 for details).

On eviction of a corrupted dirty block from LLC, the corrupted part of the block is reconstructed by querying either the owner or an elected sharer depending on the extended state of the block. If the block is found dirty in the private cache of the owner, the entire block is sent to the LLC, as usual. If the extended coherence state of the block is shared, one of the sharers is elected on-the-fly to supply the corrupted part of block, and other sharers are back-invalidated. The back-invalidation request to the elected sharer indicates which part (either the first $4 + \lceil \log_2(C) \rceil$ bits or the first $4 + C$ bits) of the block is needed

[6] Our simulated system implements sequential consistency and does not support eager-exclusive responses [3, 33].

for reconstruction. The elected sharer invalidates its copy of the block after supplying the requested corrupted part of the block with the back-invalidation acknowledgment.

In summary, the in-LLC coherence tracking mechanism can significantly delay the completion of reads to shared blocks. Additionally, this mechanism generates a slightly higher volume of interconnect traffic compared to a baseline system that is provisioned with a sufficiently large sparse directory. We quantify these two performance aspects of this mechanism in the next section.

The LLC needs to execute additional writes to the data array for updating the coherence state. These writes are, however, off the critical path and the LLC has ample free write bandwidth to handle these. Also, the coherence action (if any) for a block in the corrupted state (V=0, D=1) can be initiated only after the data block is read out and the first few bits are examined. However, this few cycles of additional delay in initiating the coherence actions for a subset of the shared accesses constitutes a very small percentage of the overall round-trip latency of a private cache miss for the scale of the systems we are dealing with. As a result, this additional delay has negligible impact on the overall performance.

4.3.3 Performance Analysis

The in-LLC coherence tracking technique suffers from two shortcomings. First, the read accesses to blocks in corrupted shared state require three transactions in the critical path compared to two transactions in the baseline sparse directory. This can be a major performance concern. Second, the reconstruction of the LLC blocks introduces small additional network traffic in the form of the first few bits ($4 + \lceil \log_2(C) \rceil$ or $4 + C$) of a subset of the blocks evicted from the private cache hierarchy.

Figure 4.4 quantifies the execution time of the in-LLC coherence tracking mechanism normalized to a 2× sparse directory. For each application, we evaluate two implementations. The left bar corresponds to a storage-heavy implementation where each LLC block's tag is extended to track coherence. The right bar corresponds to the in-LLC tracking mechanism that we introduced in Sections 4.3.1 and 4.3.2. According to Table 4.1, the number

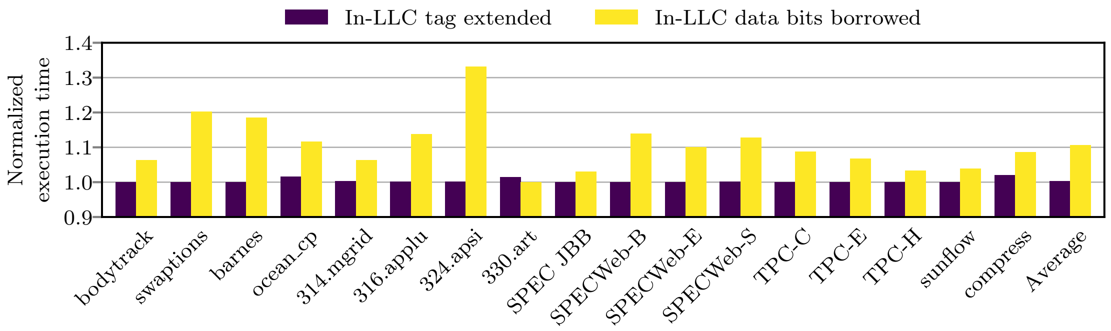

Figure 4.4: Performance of in-LLC coherence tracking normalized to a 2× sparse directory.

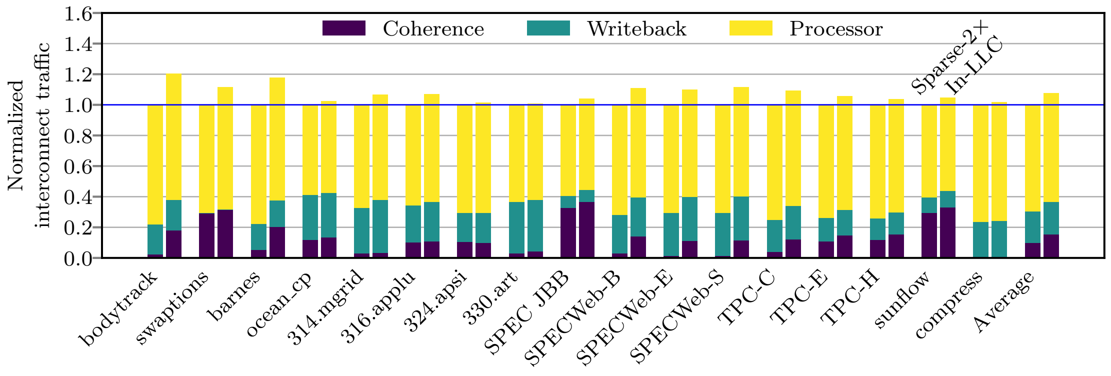

Figure 4.5: Interconnect traffic for in-LLC coherence tracking normalized to a 2× sparse directory.

of blocks in the LLC is same as the number of entries in a 2× sparse directory. As a result, the storage-heavy implementation delivers similar average performance as the baseline 2× sparse directory. On the other hand, the in-LLC tracking mechanism introduced in this chapter suffers from an 11% increase in execution cycles, on average. Several applications suffer from more than 10% increase in execution time. For each application, the difference in performance between the two bars arises from the lengthened critical path (three-hop) of the read requests to blocks in the shared corrupted state in the in-LLC tracking mechanism that borrows data bits to maintain coherence information. In the following, we study the performance of this in-LLC coherence tracking mechanism in more detail.

Figure 4.5 quantifies the interconnect traffic (in bytes) of the in-LLC tracking mechanism normalized to the 2× sparse directory baseline. For each application, the left bar corresponds to the 2× sparse directory baseline and the right bar corresponds to the in-

LLC tracking mechanism that borrows data bits to maintain coherence information. Each bar is divided into three segments representing three different types of messages. The private cache misses and their responses constitute the processor messages. The eviction notices from the cores and their acknowledgements constitute the writeback messages. The forwarded requests from the home LLC bank and the corresponding busy-clear messages (if any) coming back to the home LLC bank constitute the coherence messages. On average, the processor and writeback traffic increases by about a percentage each in the in-LLC tracking mechanism. The processor traffic increases due to an increased volume of negative acknowledgements and retries arising from a larger number of LLC blocks being in the busy state waiting to complete forwarded shared read requests. The writeback traffic increases due to inclusion of the first few bits of the evicted block required for LLC block reconstruction in some cases. The coherence traffic increases by more than 5%, on average. This increase is primarily due to the extra forwarded requests arising from the reads to the shared corrupted blocks.

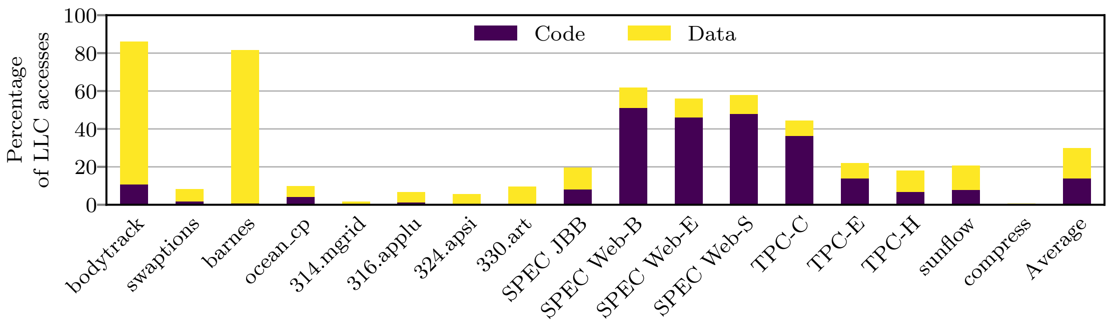

Figure 4.6: Percentage of LLC accesses which suffer an increase in critical path.

Figure 4.6 shows, for each application, the percentage of the LLC accesses that require a three-hop transaction in the in-LLC protocol, while the baseline 2× sparse directory could have served these through two-hop transactions. These are essentially read accesses to blocks in the shared corrupted state. On average, 30% of LLC accesses suffer from an increase in the critical path. For some of the commercial applications, among the lengthened accesses, the code accesses are more in population than the data accesses (see SPEC Web, TPC-C, and TPC-E). For bodytrack, barnes, and SPEC Web, more than

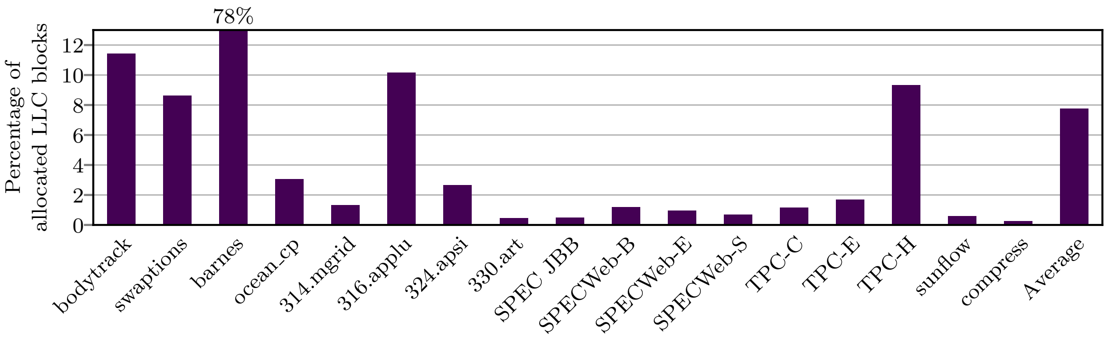

Figure 4.7: Percentage of allocated LLC blocks which experience lengthened accesses.

50% of the LLC accesses suffer from an increased critical path. For scientific and general purpose applications more data accesses suffer increase in critical path than code accesses. Figure 4.7 further shows, for each application, the percentage of the allocated LLC blocks which experience these lengthened accesses. These blocks are a subset of those shown in Figure 4.2. On average, only 8% of the allocated LLC blocks are enough to cover all the offending accesses. Barnes is a clear outlier with 78% of the blocks experiencing lengthened accesses. Among the rest, only bodytrack, swaptions, 316.applu, and TPC-H have more than 5% LLC fill population experiencing accesses with lengthened critical path. This result clearly points to a viable sparse directory design that can track this small fraction of LLC blocks and eliminate the performance drawback of the in-LLC protocol. This observation forms the foundation of our tiny directory proposal.

To further understand the extent of sharing experienced by the blocks considered in Figure 4.7, we introduce the **S**hared **T**hree-hop **R**ead **A**ccess (STRA) ratio. The STRA ratio for an allocated LLC block is the fraction of read accesses to the block which need to be forwarded to a sharer because the state of the block is shared corrupted. All blocks considered in Figure 4.7 have non-zero STRA ratios and all other blocks have zero STRA ratio. We classify the blocks with non-zero STRA ratios into seven categories $C_1, \ldots, C_7$. The category C_i for $i \in [1, 6]$ includes all LLC blocks with STRA ratio $\in (1 - \frac{1}{2^{i-1}}, 1 - \frac{1}{2^i}]$. The category C_7 includes all the LLC blocks with STRA ratio $\in (1 - \frac{1}{64}, 1]$. Figure 4.8 shows the distribution of the allocated LLC blocks with non-zero STRA ratios. Figure 4.9 shows the distribution of the LLC read accesses to shared corrupted blocks based on the category of the involved block. On average, we see that categories C_6 and C_7 account for 54% of these LLC accesses (please refer to the Average bar in Figure 4.9), while these two categories cover only 12% of the LLC blocks that source the offending accesses (please refer to the Average bar in Figure 4.8). Combining Figures 4.7, 4.8, and 4.9, we conclude that by tracking only 12% of 8% i.e., about 1% of all allocated LLC blocks in a small sparse directory, we can bring down the critical path of about 54% offending LLC accesses; to achieve this, the sparse directory hardware needs to focus on the blocks of C_6 and C_7 categories only. This observation further substantiates the possibility of a tiny directory, which can track the coherence information of a small fraction of blocks.

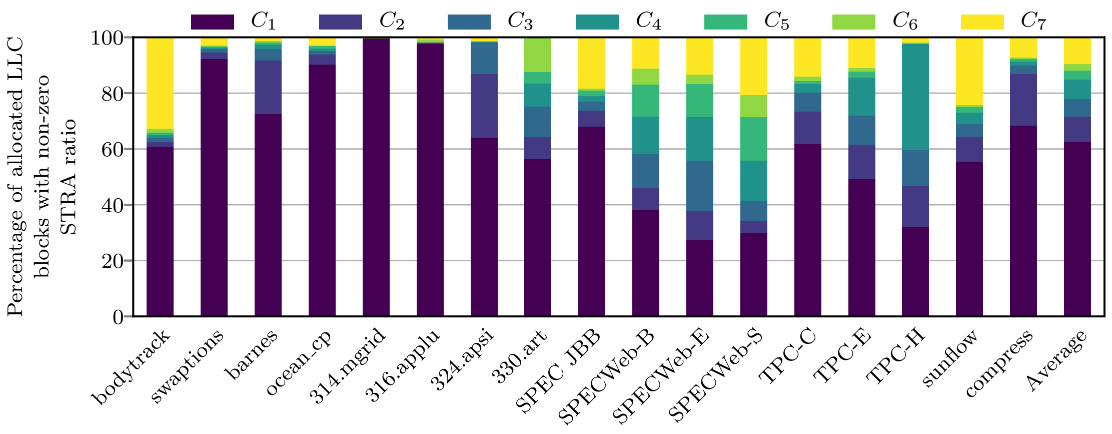

Figure 4.8: Distribution of the allocated LLC blocks based on the STRA ratio.

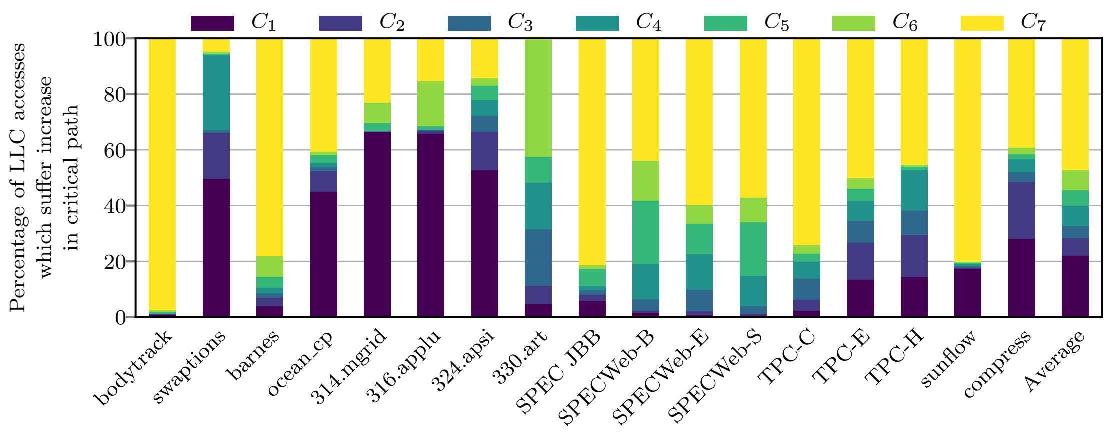

Figure 4.9: Distribution of offending LLC accesses based on the accessed block category.

4.4 Tiny Directory Proposal

The tiny sparse directory augments the in-LLC coherence tracking mechanism. The goal of the tiny directory design is to track the coherence information pertaining to a subset of blocks with high STRA ratio. Once tracked in the directory, this subset of LLC blocks will no longer be in the corrupted state and therefore, the read requests to this subset can be responded to through two-hop transactions as in the baseline sparse 2× directory. However, the small size of the tiny directory makes the selection of the blocks that are tracked in the directory very important. If too many blocks are admitted into the tiny directory, the directory entries will be prematurely evicted without offering any improvement in performance. There are two situations in which a block can be considered for being tracked in the tiny directory: (i) when a read request comes for a block which is in the corrupted state, and (ii) when an instruction read request comes for a block in unowned/non-shared/invalid state. As already noted, instruction reads are always responded to in the shared state to accelerate code sharing. In both these situations, if the block is tracked in the tiny directory, the subsequent shared read requests to such a block can be concluded using two-hop transactions.

The tiny directory design consults an allocation policy in these two situations to decide if the requested block's coherence information should be tracked in the tiny directory. If the decision is not to allocate a tiny directory entry for the block, the in-LLC coherence tracking extensions, discussed in Section 4.3.2, are used to track coherence information for the block. On the other hand, if the decision is to allocate a tiny directory entry for tracking the requested block, the LLC block is reconstructed (in case it is in a corrupted state) by forwarding the request to an elected sharer or the owner and asking the elected sharer or the owner to not only forward the block to the requester but also send the corrupted bits of the block to the LLC. The LLC block is switched to a non-corrupted valid state. The coherence state and sharer information of the block are transferred to the allocated tiny directory entry for further tracking.

On eviction of a tiny directory entry, instead of back-invalidating the sharers, the evicted entry's coherence state and tracking information are transferred to the corresponding LLC data block and the LLC block transitions to an appropriate corrupted state. If the evicted entry's data block is not present in the LLC (such cases are rare), the sharers are back-invalidated. For the best outcome, it is important to carry out judicious allocations in the tiny directory and minimize the number of pre-mature evictions. We explore two allocation/eviction policies next.

4.4.1 Selective Allocation Policies

The selective allocation policies make use of the STRA ratio that the LLC blocks would have experienced in the in-LLC coherence tracking mechanism. In addition to the seven categories ($C_1, \ldots, C_7$) of non-zero STRA ratio, we use C_0 to denote the category of blocks with zero STRA ratio. For estimating the STRA ratio of an LLC block, two six-bit saturating counters, namely STRA Counter (STRAC) and **O**ther **A**ccess **C**ounter (OAC), are maintained for the block. The STRAC is incremented on LLC read accesses which find the block being requested in the shared state (such an access would have resulted in a three-hop critical path in in-LLC coherence tracking). The OAC is incremented on all other LLC accesses (except writeback) to the block. Both the counters of the block are halved when any of the counters has saturated. The STRA ratio estimate for the block is given by the fraction $\frac{STRAC}{STRAC+OAC}$. For the blocks being tracked in the tiny directory, the directory entry is extended by twelve bits to accommodate the two counters. For the LLC blocks in corrupted state, twelve bits are borrowed from the LLC data block to maintain the two counters (this lengthens the corrupted portion by twelve more bits). When the coherence information is transferred between a tiny directory entry and the corresponding LLC data block, both the access counters are also transferred. Once a block returns to the unowned/non-shared state, the counters are reset and the STRA ratio for the block is deemed zero. In the following, we discuss two allocation/eviction policies for the tiny directory. Note that given the STRA ratio of a block, the hardware can easily determine the STRA category of the block.

Dynamic STRA Policy

The Dynamic STRA (DSTRA) policy first looks for an invalid way in the tiny directory target set. If such a way is present in the target set, it decides to utilize the invalid way to track coherence information for the requested block. If there is no such way, it locates the way w with the lowest STRA category (say, C_i) in the target set. If there are multiple ways with the lowest STRA category, the one with the lowest physical way id is selected. Let the STRA category of the block B for which the tiny directory allocation policy is invoked be C_j. The DSTRA policy victimizes the entry w (which is currently tracking a block belonging to STRA category C_i) to track block B only if $i < j$. On the other hand, if $i \geq j$, the DSTRA policy denies tracking the block B in the tiny directory. In summary, this policy tries to track a subset of blocks with maximum STRA ratio in the tiny directory. However, one major shortcoming of this policy is that a block belonging to the C_7 STRA category, once tracked in the tiny directory, will occupy a tiny directory entry for a long time until its STRA ratio comes down. This becomes particularly problematic if the block is not accessed for a long time. This may unnecessarily lower the overall utilization of the tiny directory entries. Our next policy proposal remedies this problem.

DSTRA with Generational NRU Policy

The DSTRA with generational not-recently-used policy (DSTRA+gNRU) divides the entire execution into intervals or generations. Each tiny directory entry is extended with two state bits, namely, a reuse (R) bit and an eviction priority (EP) bit. When a tiny directory entry is filled or accessed, the R bit of the entry is set and the EP bit is reset, recording the fact that the entry has been recently accessed and must not be prioritized for eviction in the current interval. At the end of each interval, the tiny directory entries are examined and if an entry's R bit is reset, its EP bit is turned on signifying that the entry can be considered for eviction in the next interval. The R bits of all entries are gang-cleared at the beginning of each interval signifying the start of a new generation of reuses.

The DSTRA+gNRU policy proceeds similarly to the DSTRA policy and selects a way

w with the lowest STRA category (C_i) in the target set. If there are multiple ways with the lowest STRA category, the ones with their EP bits set are selected and then among them the one with the lowest physical way id is selected. Let the STRA category of the block B for which the tiny directory allocation policy is invoked be C_j. The DSTRA+gNRU policy victimizes the entry w (which is currently tracking a block belonging to STRA category C_i) to track block B only if one of the two following conditions is met: (i) $i < j$, (ii) $i == j$ and the EP bit of w is set. The second condition effectively creates an avenue for replacing useless entries of a certain STRA category. If none of the conditions are met, the DSTRA policy denies tracking the block B in the tiny directory.

The length of a generation needs to be chosen carefully. If a generation is too short, it may fail to capture important reuses. We set the generation length to the interval length between two consecutive reuses to a tiny directory entry, averaged across all entries. We dynamically estimate this interval length as follows. The interval length is measured in multiples of 4K cycles and the maximum interval length that our hardware can measure is 4M cycles. Each tiny directory slice attached to an LLC bank maintains a ten-bit counter T which is incremented by one every 4K cycles (measured using a twelve-bit counter). Each tiny directory entry is extended by ten bits to record the value of the counter T whenever the entry is accessed. On an access to a tiny directory entry, the last recorded value of counter T in the tiny directory entry (call it T_{last}) is compared with the current value of counter T in the slice (call it $T_{current}$). If $T_{last} < T_{current}$, the difference between $T_{current}$ and T_{last} is added to a counter A. The counter A is maintained per tiny directory slice and records the accumulated time between two consecutive accesses to a tiny directory entry. Another counter B maintained per tiny directory slice records the number of values added to counter A. At any point in time, the generation length used by a tiny directory slice is estimated as $\frac{A}{B}$. At the beginning of an interval, this value is copied to a generation length counter, which is decremented by one every 4K cycles. A generation ends when this counter becomes zero. Both the counters A and B are halved when either of them has saturated. When counter T saturates, it is reset to zero.

4.4.2 Introducing Robustness: Spilling into LLC

The tiny directory is only capable of identifying and tracking the coherence state of a subset of blocks that are most likely to suffer in terms of lengthened critical path of shared read accesses in the in-LLC coherence tracking mechanism. Since the size of this performance-critical shared working set of an application is not known beforehand and may even vary during execution, it is impossible to design an optimally-sized tiny directory that can offer robust and reliable performance for a wide range of applications. To address this problem, we augment the tiny directory design with the option of selectively spilling a subset of coherence tracking entries into the LLC. This option is particularly helpful when the tiny directory is too small to track the critical shared working set. A spilled coherence tracking entry occupies a tag and the corresponding data block in the LLC. It is different from the data block for which coherence is being tracked. As a result, a fundamental challenge in enabling a coherence tracking entry spill policy is to ensure that the volume of LLC misses does not increase due to the pressure of the spilled tracking entries. We first discuss the organization and maintenance of a spilled coherence tracking entry. Next, we describe the selective spill policy, which identifies the coherence tracking entries eligible for spilling.

Organization of Spilled Entries

The reason for enabling spilling of coherence tracking entries into the LLC is to avoid lengthening the critical path of shared read accesses when the tiny directory is unable to track all such shared blocks. As a result, a coherence tracking entry E_B of a block B can be spilled into the LLC only if B is currently in the shared state. A spilled coherence tracking entry E_B is allocated in a way in the same set as block B. Since B and E_B have the same tag, this allocation decision guarantees that in an LLC set, there can be at most two tag matches on a lookup. On a lookup, when there is no tag match, it is an LLC miss; when there is exactly one tag match, the tag match is guaranteed to be for a block and not for a spilled directory entry. When there are two tag matches, it is necessary to distinguish between the actual block from its spilled directory entry. To distinguish between the block B and the block E_B holding its spilled directory entry, we use the state

(V=0, D=1) for the spilled tracking entries. B cannot be in a corrupted state when it has a directory entry (spilled or otherwise) and hence, it will have V=1. The LLC replacement policy always victimizes a spilled coherence tracking entry E_B before the corresponding block B because victimizing B from the LLC can be far more expensive for a future access than victimizing E_B. This is ensured by the LRU position update policy of the LLC. Whenever B is accessed, E_B is also accessed for updating the tracking information of B. Whenever these two blocks are accessed, we move E_B to the MRU position first and then move B to the MRU position of the LLC set. When E_B is chosen as a victim, the coherence information is transferred to B and B switches to the corrupted shared state.

If an LLC lookup indicates two tag matches, we know that the one with state V=1 corresponds to the data block and the other one is the spilled coherence tracking entry for the block. On the other hand, if the lookup returns a single tag match, the state of the matched tag decides if the block is in a corrupted state (V=0, D=1) or not (V=1). As usual, the tiny directory is always looked up in parallel with the LLC and a tiny directory hit indicates that the coherence information of the block is being tracked in the tiny directory.

On an access to a data block B, if the coherence tracking entry E_B is also in the same set, the two blocks have to be read out sequentially. To avoid lengthening the critical path, on a read request, we read out the data block B first and respond to the requester (recall that spilling is allowed only for blocks in the shared state). Next, we read out E_B and update the coherence tracking information. Finally, we move E_B to the MRU position and then move B to the MRU position within the LLC set. On a read-exclusive request, we read out E_B first and send out the invalidations and also ask an elected sharer to forward the data block to the requester. On an upgrade request, we read out E_B first and send out the invalidations. For both read-exclusive and upgrade requests, the block E_B is invalidated and the coherence information is transferred to B, which now switches to the corrupted exclusive state. This is because maintaining a spilled directory entry is beneficial only for blocks in the shared state.

Selective Spill Policy

The selective spill policy for coherence tracking entries determines if the coherence information of a block can be tracked by spilling it in the LLC. This policy is invoked in two situations: (i) when the tiny directory's allocation policy declines to track the coherence information of a requested block in the tiny directory, and (ii) eviction of a tiny directory entry corresponding to a block in the shared state. If the policy decision is not to spill in the LLC, the in-LLC coherence tracking extensions are used to track the coherence information of the involved block. If the policy decision is to spill in the LLC, a way is allocated in the same LLC set as the involved block to track the coherence information of the block. In this case, if the involved block is found in a corrupted state in the LLC, it is reconstructed following the reconstruction procedure discussed already and the block transitions to a non-corrupted valid state (V=1).

Whenever the spill policy is invoked, its goal is to allow spilling of coherence tracking entries for blocks with high STRA ratio. At the same time, the spill policy must keep a check on the LLC miss rate for data blocks. Let C_j be the STRA category of the block which is trying to spill its coherence tracking entry in the LLC under one of the two aforementioned situations when the spill policy is invoked. We formulate the selective spill policy design problem as follows. The selective spill policy should dynamically determine the highest STRA category C_i such that the coherence tracking entries for the blocks with STRA category C_j with $j \geq i$ can be spilled in the LLC whenever needed while guaranteeing that the LLC miss rate for data blocks increases by no more than a predefined value of δ. The value δ represents the tolerance limit for LLC miss rate. Each LLC bank independently implements this policy and determines a suitable C_i for the bank. The index i of this computed C_i for an LLC bank will be referred to as the STRA spill threshold category index of the bank and this selective spill policy will be referred to as the Dynamic Spill policy. We discuss its implementation in the following.

In each LLC bank, sixteen sets are kept aside that do not admit any spilled coherence tracking entries. These sets are used to estimate the LLC bank's miss rate without spilling ($MR_{no-spill}$). The remaining sets exercise spilling for STRA categories C_j such that $j \geq i$, given a dynamically computed STRA spill threshold category index i for the LLC bank. From these sets, the LLC bank's miss rate with spilling (MR_{spill}) can be determined. We define a window of observation for an LLC bank as 8K accesses (except write-backs) to the bank. At the end of each observation window, if $MR_{spill} \leq MR_{no-spill} + \delta$ is satisfied (meaning that due to spilling, the LLC bank's miss rate increases by no more than δ), the STRA spill threshold category index i is decreased by one in that bank so that a bigger volume of spills can be admitted in the next observation window. On the other hand, if $MR_{spill} \leq MR_{no-spill} + \delta$ is not satisfied, i for the bank is increased by one so that the spill volume can be reduced. We note that the value of i saturates at zero and seven on the two sides of the admissible range.

The aforementioned policy for dynamically determining the STRA spill threshold category index may lead to oscillations in the index value around the convergence point unless the index i saturates to zero or seven. Such oscillations are easy to detect and the STRA spill threshold category index can be fixed to one of the two oscillation values such that $MR_{spill} \leq MR_{no-spill} + \delta$ is satisfied. However, fixing the index value to avoid oscillation may cause the state of the algorithm to get stuck at that index value leading to lost opportunity of spilling more in certain phases of execution. Coming out of such a state will require complex mechanisms to detect phase changes when a new lower index value can be tried. This is complicated by the fact that MR_{spill} for a certain STRA spill threshold category index cannot be determined by sampling a few LLC sets (like the way we determine $MR_{no-spill}$) because the spill volume distribution is non-uniform and highly skewed toward the LLC sets that accommodate shared blocks. We, however, note two important aspects about this oscillation. First, if an oscillation at all happens, it is restricted to the few phases of execution that experience high to moderate volumes of spilling because small amount of spilling cannot change the LLC miss rate much. Second, since the length of the observation window is quite large (8K accesses per bank $\times$ 128 banks or 1M LLC accesses on average), the oscillation in the index value happens at a reasonably slow rate. At the end, to keep the design simple, we decide to use our Dynamic Spill policy without any change to arrest oscillation. Our evaluation of this policy shows that even with the possibility of such oscillations in certain phases, the increase in the LLC miss rate due to spilling never exceeds the guarantee offered by the value of δ.

Selection of an appropriate δ is important for the success of the proposed spill policy. We define the overall STRA ratio of an application as the number of LLC reads to blocks in the shared state over the total number of LLC accesses (except writebacks). In general, if an application has a very low LLC miss rate, it may not be able to tolerate a large increase in LLC miss rate because such applications are typically very latency-sensitive. On the other hand, if an application is undergoing a phase of overall high STRA ratio, it may be possible to convert a larger proportion of LLC hits to misses and gain in terms of shared read hit latency by spilling more. Within each LLC bank, we measure the miss rate and the overall STRA ratio. At the end of each window of observation, each LLC bank independently classifies the running application into four possible categories: (A) LLC bank's miss rate is at least 10% and STRA ratio is at least 0.4, (B) LLC bank's miss rate is at least 10% and STRA ratio is below 0.4, (C) LLC bank's miss rate is below 10% and STRA ratio is at least 0.4, and (D) LLC bank's miss rate is below 10% and STRA ratio is below 0.4. At the beginning of each observation window, each LLC bank independently decides the value of δ to be used in that bank depending on the category of the application observed during the last window: $\delta_A = \frac{1}{4}$, $\delta_B = \frac{1}{32}$, $\delta_C = \frac{1}{16}$, $\delta_D = \frac{1}{32}$. The categories with higher STRA ratio are assigned higher δ values (higher tolerance for LLC miss rate increase) while keeping the miss rate profile in mind. These values may require tuning depending on the system configuration.

4.4.3 Coherence Processing Latency at LLC

Among the coherence processing paths traversed by the tiny directory proposal at the LLC bank controller, there are two situations where the critical path gets slightly lengthened compared to the baseline. Both the cases arise from accessing a block in the corrupted state. If the accessed block is in the corrupted shared state, the LLC tag and data must be accessed serially followed by decoding of the coherence state from the data block before responding to the requester. In the baseline, the critical path through the LLC bank controller for accessing such a block would involve only the serial access of the LLC tag and data (overlapped with sparse directory access). In this case, we charge one extra cycle of LLC latency for the tiny directory implementation accounting for the coherence state decoding overhead. If the accessed block is in the corrupted exclusive state, the tiny directory proposal must access the LLC tag and data serially and then decode the coherence state before forwarding the request to the owner. In the baseline, the critical path through the LLC bank controller for accessing such a block would involve only the LLC tag access latency overlapped with the sparse directory lookup latency. In this case, the tiny directory proposal suffers from two additional cycles of LLC data access latency (see Table 4.1) followed by one cycle of coherence state decoder latency. We model all these additional latencies in our evaluation.

4.4.4 Coherence Protocol Complexity

The tiny directory protocol is extended from the baseline MESI protocol and introduces two new coherence states, namely corrupted shared and corrupted exclusive. The corrupted exclusive state combines corrupted M and corrupted E states. The new protocol state machine includes the transitions into and out of these two new states. Since only two new states have been added, we believe that the additional complexity is small and the additional verification effort is within the reasonable margins.

4.5 Simulation Results

We evaluate our proposal in this section for four different tiny directory sizes: $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$. The $\frac{1}{32}\times$ and $\frac{1}{64}\times$ sizes have respectively 64 and 32 entries per tiny directory slice attached to an LLC bank. Both these sizes exercise eight-way set-associative directory slices. The $\frac{1}{128}\times$ and $\frac{1}{256}\times$ sizes have respectively 16 and 8 entries per tiny directory slice and exercise fully-associative configuration for each slice. Each directory entry has a size of 155 bits (128-bit sharer vector, 12 bits for STRAC and OAC, 10 bits for the timestamp counter used to estimate the generation length in the gNRU policy, two bits for R and EP states used by the gNRU policy, one bit for pending/busy transient state, and two coherence state bits for tracking invalid, exclusively owned and shared states). Additionally, each directory entry has a tag of the following lengths (we assume 48-bit physical address): 32 bits for $\frac{1}{32}\times$, 33 bits for $\frac{1}{64}\times$, 35 bits for $\frac{1}{128}\times$ and $\frac{1}{256}\times$. As a result, the total storage investment for coherence tracking across all 128 slices is as follows: 187 KB for $\frac{1}{32}\times$, 94 KB for $\frac{1}{64}\times$, 47.5 KB for $\frac{1}{128}\times$, and 23.75 KB for $\frac{1}{256}\times$.

Figures 4.10 and 4.11 evaluate our proposal for $\frac{1}{32}\times$ and $\frac{1}{64}\times$ sizes, respectively. These figures quantify the percentage increase in execution cycles compared to a $2\times$ directory. For each tiny directory size, we show the results with the DSTRA allocation policy, DSTRA+gNRU allocation policy, and DSTRA+gNRU augmented with dynamic spilling (DynSpill) of coherence tracking entries. For the $\frac{1}{32}\times$ size (Figure 4.10), both DSTRA and DSTRA+gNRU policies are, on average, within 1% of the performance of $2\times$ directory; when dynamic spilling is enabled, the gap reduces to 0.5%. Referring back to Figure 4.4, we note that the in-LLC coherence tracking mechanism is 11% worse than the $2\times$ directory. Introduction of a tiny directory bridges this gap.

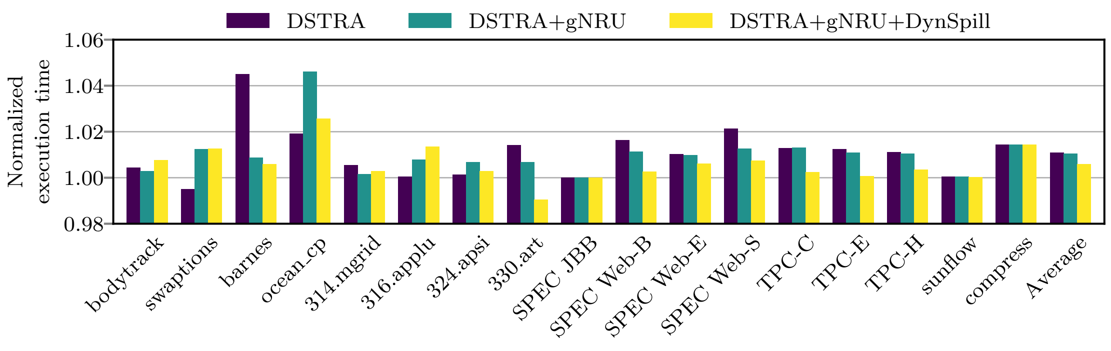

Figure 4.10: Performance of $\frac{1}{32}\times$ tiny directory normalized to a sparse $2\times$ directory.

For the $\frac{1}{64}\times$ size (Figure 4.11), the gNRU-assisted allocation policy begins to gain in importance in some of the applications (ocean_cp and SPECWeb). On average, the DSTRA policy has 3% higher execution cycles compared to the $2\times$ directory, while the DSTRA+gNRU policy is only 2% away from the $2\times$ directory. Dynamic spilling further brings this gap down to 1%. Dynamic spilling is particularly helpful for 324.apsi, SPECWeb, and TPC. Dynamic spilling, however, hurts performance by a couple of per-

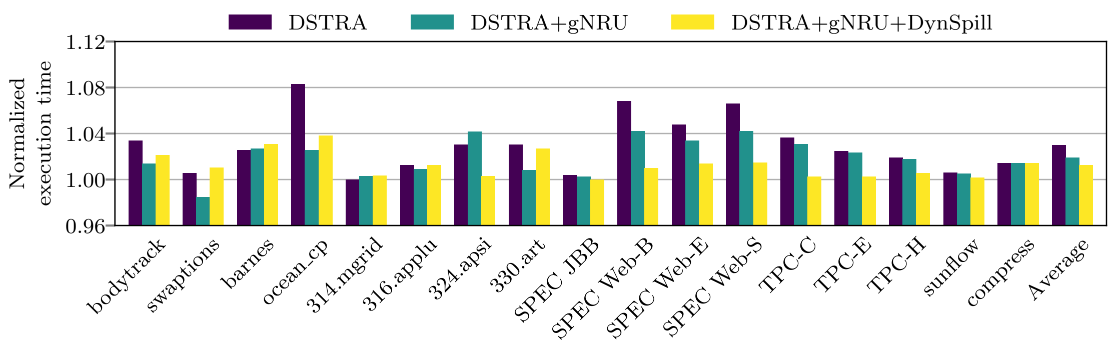

Figure 4.11: Performance of $\frac{1}{64}\times$ tiny directory normalized to a sparse $2\times$ directory.

centages in swaptions and 330.art due to LLC contention.

Referring back to the discussion related to Figure 4.3, we note that a skew-associative directory that tracks only shared blocks suffers from a 12% slowdown for the $\frac{1}{64}\times$ size compared to a $2\times$ directory, on average. Our set-associative tiny directory without dynamic spilling at this size performs far better underscoring the success of the DSTRA and the DSTRA+gNRU policies which capture a critical subset of the shared blocks. Referring back to Figure 4.7, we note that this critical subset accounts for 78% of all allocated LLC blocks for barnes. Even for this application, our tiny directory proposal is able to capture the instantaneous working set of these critical blocks and deliver performance close to a $2\times$ directory.

Figures 4.12 and 4.13 evaluate our proposal for $\frac{1}{128}\times$ and $\frac{1}{256}\times$ sizes, respectively. At these two sizes, the gNRU policy gains further in importance in several applications. On average, for the $\frac{1}{128}\times$ size, the DSTRA and the DSTRA+gNRU policies have 6% and 5% higher execution cycles compared to the $2\times$ directory. The dynamic spill policy assumes significant importance at these small directory sizes and brings down the gap between our proposal and the $2\times$ directory to 1%. Referring back to Figure 4.3, we note that a sparse directory that tracks only shared blocks suffers from a 28% slowdown for the $\frac{1}{128}\times$ size compared to a $2\times$ directory, on average. Our tiny directory proposal successfully wipes away this performance loss.

For the $\frac{1}{256}\times$ size (Figure 4.13), the DSTRA and the DSTRA+gNRU policies have 8% and 6% higher execution cycles compared to the $2\times$ directory, on average. Dynamic spilling reduces this gap to 1%. In summary, our tiny directory proposal offers robust performance staying within a percentage of a sparse 2× directory as the tiny directory size is varied between $\frac{1}{32}\times$ and $\frac{1}{256}\times$ (187 KB to 23.75 KB).

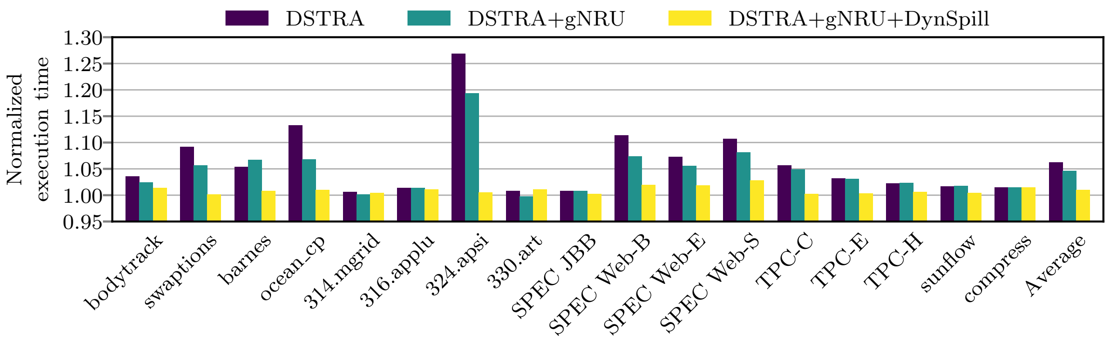

Figure 4.12: Performance of $\frac{1}{128}\times$ tiny directory normalized to a sparse 2× directory.

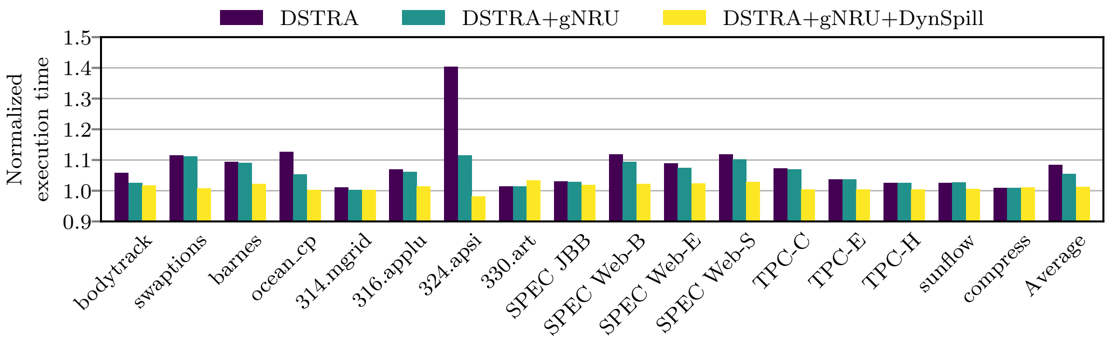

Figure 4.13: Performance of $\frac{1}{256}\times$ tiny directory normalized to a sparse 2× directory.

4.5.1 Analysis of Performance

The main purpose of the tiny directory proposal is to eliminate most of the additional three-hop transactions that the in-LLC coherence mechanism introduced. When these three-hop transactions get replaced by the two-hop transactions as in the sparse 2× directory, the performance is expected to be similar to the 2× directory. Referring back to Figure 4.6, we note that the percentage of LLC accesses that suffer from an increased critical path because they get extended to three-hop transactions in the in-LLC coherence tracking mechanism is 30% on average. To confirm that our proposal is able to address this problem successfully, Figures 4.14 and 4.15 show the percentage of the LLC accesses that suffer from an increase in the critical path for a $\frac{1}{32}\times$ tiny directory system and a $\frac{1}{256}\times$ tiny directory system, the two extreme points of our size spectrum. For a $\frac{1}{32}\times$ tiny directory, the DSTRA and the DSTRA+gNRU policies have only 3% and 2% such LLC accesses on average. The dynamic spill policy brings this average to under 1%. For a $\frac{1}{256}\times$ tiny directory, this percentage increases significantly for the DSTRA and the DSTRA+gNRU policies. These policies experience 23% and 20% such LLC accesses respectively (still lower than in-LLC mechanism), while the dynamic spill policy successfully brings this average down to only 4%. These small residual extra three-hop transactions cause a percent loss in performance.

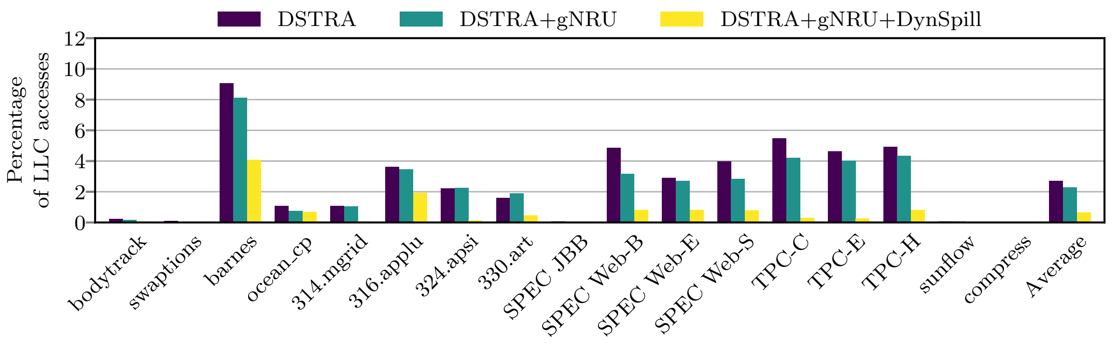

Figure 4.14: Percentage of LLC accesses which suffer from an increase in critical path in a $\frac{1}{32}\times$ tiny directory.

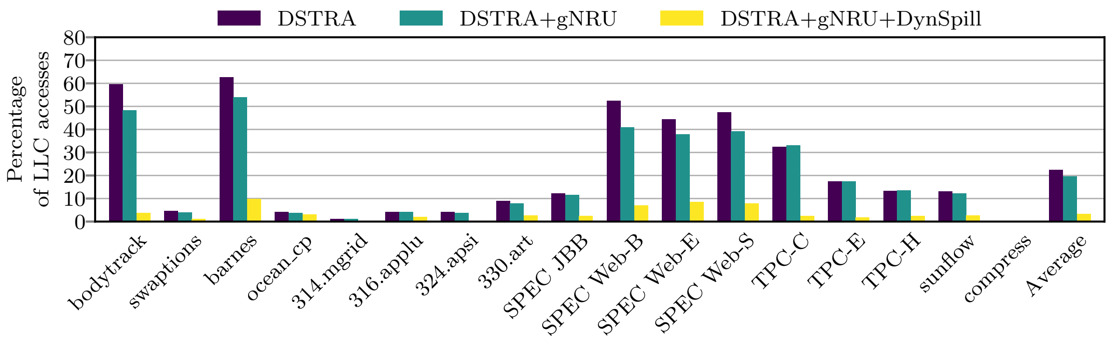

Figure 4.15: Percentage of LLC accesses which suffer from an increase in critical path in a $\frac{1}{256}\times$ tiny directory.

The success of the tiny directory in reducing the number of extra three-hop transactions depends on the number of hits that a tiny directory entry enjoys. Figure 4.16 shows the number of tiny directory hits for the DSTRA+gNRU policy normalized to the

DSTRA policy for all the four directory sizes. As the directory size decreases from $\frac{1}{32}\times$ to $\frac{1}{256}\times$, the gNRU policy gains in importance. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, the DSTRA+gNRU policy offers, respectively, 3%, 12%, 23%, and 39% more directory entry hits compared to the DSTRA policy. The biggest beneficiaries of the gNRU policy are bodytrack, swaptions, barnes, ocean_cp, 330.art, and SPECWeb. The primary advantage of the gNRU policy is that it quickly removes the useless directory entries, which the DSTRA policy would have retained for a long time. This creates room for more useful directory entries to be tracked. Figure 4.17 validates this behavior by quantifying the number of allocations in the tiny directory experienced by the DSTRA+gNRU policy normalized to the DSTRA policy for all the four directory sizes. As the directory size decreases from $\frac{1}{32}\times$ to $\frac{1}{256}\times$, the gNRU policy allows a much larger number of directory fills to take place, thereby significantly increasing the effective coverage of the tiny directory. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, the DSTRA+gNRU policy observes, respectively, 2×, 7×, 50×, and 74× more directory fills compared to the DSTRA policy. Figure 4.18 quantifies the average number of hits enjoyed by a directory entry before getting replaced for the DSTRA+gNRU policy. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, this number is 59.5, 46.1, 16.6, and 17.5, respectively. This result confirms that the directory entries tracked by the DSTRA+gNRU policy are indeed important. They enjoy a significant number of hits before getting replaced even for the smallest size.

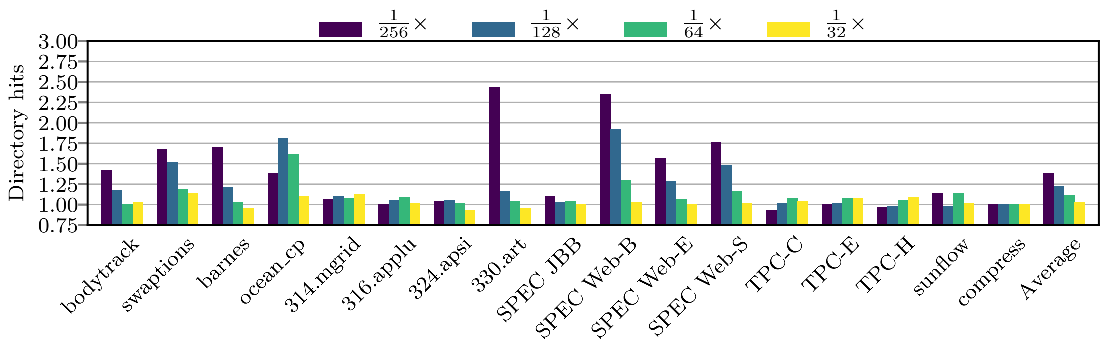

Figure 4.16: Hits in tiny directory with the DSTRA+gNRU policy normalized to the DSTRA policy.

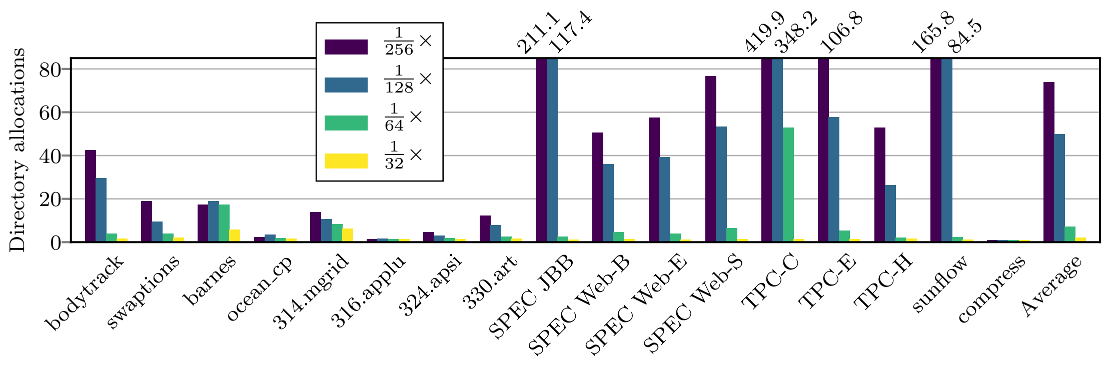

Figure 4.17: Allocations in tiny directory with the DSTRA+gNRU policy normalized to the DSTRA policy.

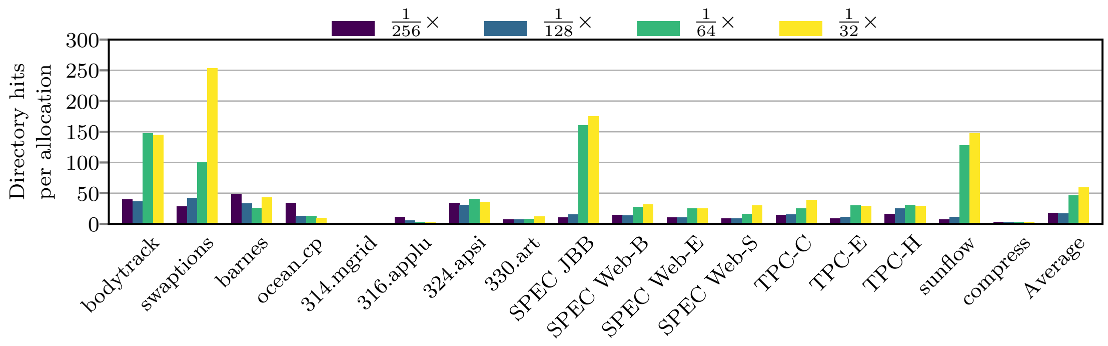

Figure 4.18: Hits per allocation in tiny directory with the DSTRA+gNRU policy.

Next, we analyze our dynamic spill policy which we have shown to be highly robust across the board. There are two aspects of the dynamic spill policy that we analyze. Figure 4.19 shows the percentage of the LLC accesses which are able to avoid increase in critical path because of spilled directory entries when using the DSTRA+gNRU+DynSpill policy. These are essentially read accesses to the blocks, the coherence tracking entries of which are spilled in the LLC. Without these spilled entries, these accesses would get extended to three-hop transactions because the data block would have been in the corrupted shared state. The percentage of such LLC accesses increases significantly as the tiny directory size drops. On average, for $\frac{1}{32}\times$, $\frac{1}{64}\times$, $\frac{1}{128}\times$, and $\frac{1}{256}\times$ directories, 2%, 5%, 11%, and 16% LLC accesses benefit from spilling. The biggest beneficiaries are bodytrack, barnes, SPECWeb, and TPC.

The second aspect of the spill policy is its influence on the LLC miss rate. We are

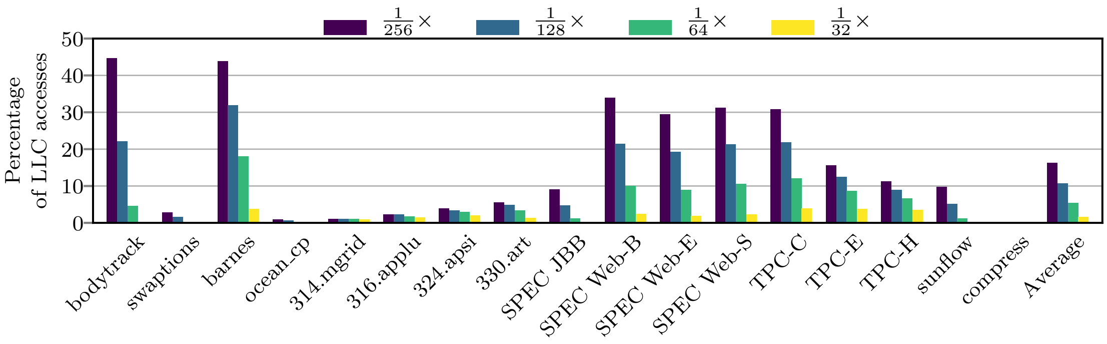

Figure 4.19: Percentage of LLC accesses which are able to avoid increase in critical path because of spilled directory entries in the LLC when using the DSTRA+gNRU+DynSpill policy.

particularly interested in the behavior of the applications that already have high LLC miss rates in the baseline. For example, the applications with more than 10% LLC miss rate include ocean_cp (35% LLC miss rate), 314.mgrid (78%), 324.apsi (12%), 330.art (63%), SPECWeb-B (14%), SPECWeb-E (19%), and SPECWeb-S (18%). Our Dynamic Spill policy guarantees an upper bound on the LLC miss rate increase through the δ values. Figure 4.20 shows the increase in LLC miss rate when using the DSTRA+gNRU+DynSpill policy relative to the sparse 2× directory. As the tiny directory size decreases, the LLC miss rate increases very slowly. Only 316.applu and 330.art show more than 1% increase in the LLC miss rate compared to the 2× directory. Across the board, the maximum increase in the LLC miss rate due to spilling is 2.1% experienced by 316.applu when operating with a $\frac{1}{256}\times$ tiny directory. We note that this is within the smallest δ (the guaranteed upper bound on LLC miss rate increase) that we use (Section 4.4.2). The average increase in the LLC miss rate is under 0.5% for all directory sizes.

To further confirm that our proposal continues to offer robust performance for smaller LLC capacities where the pressure created by the spilled directory entries can be more problematic, we evaluate our proposal in a configuration where the entire cache hierarchy is halved in terms of the number of sets (the capacity ratio between different levels is maintained) i.e., the shared LLC capacity is 16 MB in both the baseline and our proposal. In this configuration, compared to a sparse 2× directory, the DSTRA+gNRU and DSTRA+gNRU+DynSpill policies experience an average increase of 7% and 1% execution

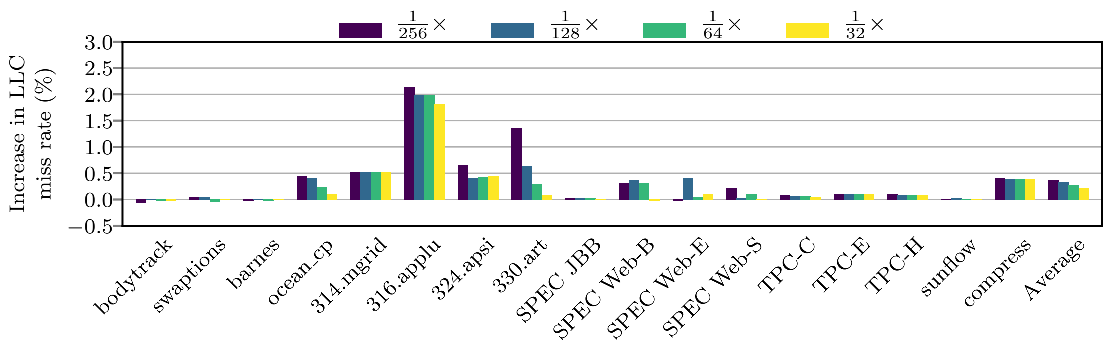

Figure 4.20: Increase in LLC miss rate due to spilling when using DSTRA+gNRU+DynSpill policy compared to a 2× sparse directory.

cycles for a $\frac{1}{128}\times$ tiny directory (eight entries fully-associative per slice) where spilling is quite prevalent.

4.5.2 Energy Comparison

We use CACTI [37] (distributed with McPAT [38]) to compute the dynamic and leakage energy consumed by the LLC and the sparse directory for 22 nm nodes. Figure 4.21 shows the dynamic, leakage, and total energy of the LLC and the sparse directory for the baseline configurations (from 2× to $\frac{1}{16}\times$) normalized to the $\frac{1}{256}\times$ tiny directory exercising the DSTRA+gNRU+DynSpill policy. Specifically, the dynamic energy of the baseline configurations is normalized to the dynamic energy of the $\frac{1}{256}\times$ tiny directory; the leakage energy of the baseline configurations is normalized to the leakage energy of the $\frac{1}{256}\times$ tiny directory; the total energy of the baseline configurations is normalized to the total energy of the $\frac{1}{256}\times$ tiny directory. We have also shown the $\frac{1}{128}\times$ tiny directory in the figure. Additionally, the figure includes the trends in the execution cycles as well. As the baseline sparse directory size decreases, the execution cycles monotonically increase, as expected. The dynamic, leakage, and total energy expense first decreases as the baseline directory size shrinks. However, beyond $\frac{1}{4}\times$ size of the baseline directory, the energy expense increases quickly due to increasing execution cycles. Compared to the $\frac{1}{256}\times$ tiny directory, the baseline dynamic energy is much lower. The extra dynamic energy consumption in the tiny directory arises primarily from the additional LLC writes that need to be done to update the coherence information in the corrupted and the spilled entries. On the other hand, the leakage and total energy expense is much lower in the tiny directory due to the drastically reduced size of the directory. The data array of a 2× directory is 8 MB in capacity compared to 47.5 KB and 23.75 KB total sizes of the $\frac{1}{128}\times$ and $\frac{1}{256}\times$ tiny directories. Overall, compared to the baseline sparse 2× directory, our proposal saves 17% and 16% of total LLC and directory energy for the $\frac{1}{128}\times$ and $\frac{1}{256}\times$ tiny directory sizes, respectively. The baseline $\frac{1}{4}\times$ sparse directory configuration comes closest to the $\frac{1}{256}\times$ tiny directory in terms of total energy expense (4% more than the $\frac{1}{256}\times$ tiny directory), but requires 1 MB space for its directory data array and performs 2.5% worse than the $\frac{1}{256}\times$ tiny directory.

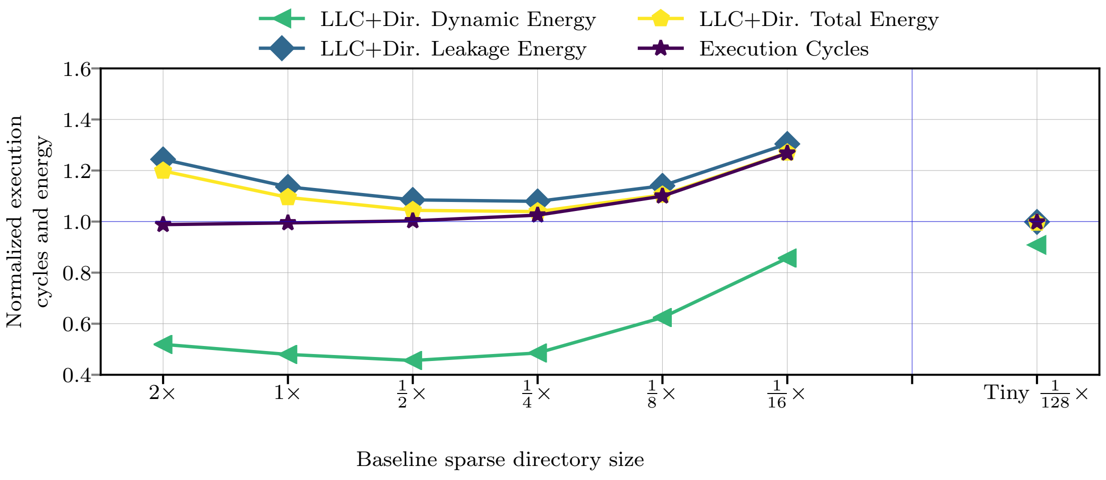

Figure 4.21: Execution cycles and energy normalized to the $\frac{1}{256}\times$ tiny directory exercising the DSTRA+gNRU+DynSpill policy.

4.5.3 Comparison to Related Proposals

Recent proposals have tried to reduce the number of sparse directory entries by addressing the overhead of tracking the private blocks. These contributions were reviewed in Section 4.1. By comparing Figure 4.3 with Figures 4.10, 4.11, and 4.12, we have already shown that our proposal performs much better than a sparse directory that tracks only shared blocks. None of the recent proposals that try to reduce the overhead of tracking the private blocks can perform better than the ideal sparse directory that tracks only shared blocks. Nonetheless, for completeness, we evaluate the state-of-the-art multi-grain directory (MgD) [86] and the Stash directory [26]. The MgD invests just one directory entry for a private region of size 1 KB, thereby saving significantly on the overhead of tracking the private blocks. The Stash directory does not track private blocks after the corresponding directory entries are evicted. As a result, on evicting a Stash directory entry that was tracking a private block, it does not generate any back-invalidation. Such a block is identified by a single bit attached to each LLC block. Later if such an LLC block receives a request from a core, the proposal resorts to broadcast to reconstruct the directory entry. Figure 4.22 evaluates a skew-associative MgD for four sizes ($\frac{1}{8}\times$, $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$) and an eight-way set-associative Stash directory for $\frac{1}{32}\times$ size. Compared to a 2× sparse directory, the MgD proposal suffers from a 0.1%, 8%, 29%, and 63% increase in average execution cycles for $\frac{1}{8}\times$, $\frac{1}{16}\times$, $\frac{1}{32}\times$, and $\frac{1}{64}\times$ sizes, respectively. The Stash directory at $\frac{1}{32}\times$ size performs 41% worse than the 2× directory on average. We find that the Stash directory is able to save a significant volume of private cache misses because it does not back-invalidate the private blocks on sparse directory eviction. However, the broadcast traffic becomes a major bottleneck in this proposal, particularly for the scale of the systems we are considering. In comparison, our proposal exercising directory sizes between $\frac{1}{256}\times$ and $\frac{1}{32}\times$ performs within 1% of a 2× sparse directory.

4.6 Summary

We have presented a novel design to track coherence information within a CMP. The design allows us to significantly scale down the traditional sparse directory size. Our proposal has three major components. First, it tracks the private block owner by borrowing a few bits of the last-level cache data block. Second, it employs a tiny directory that tracks the coherence information of a critical subset of the blocks belonging to the shared working set. Third, if the tiny directory is too small to track all the critical shared blocks, the proposal employs dynamic selective spilling of the coherence tracking entries into the LLC. Any remaining block is tracked by borrowing a few bits of the LLC data block. The simulation

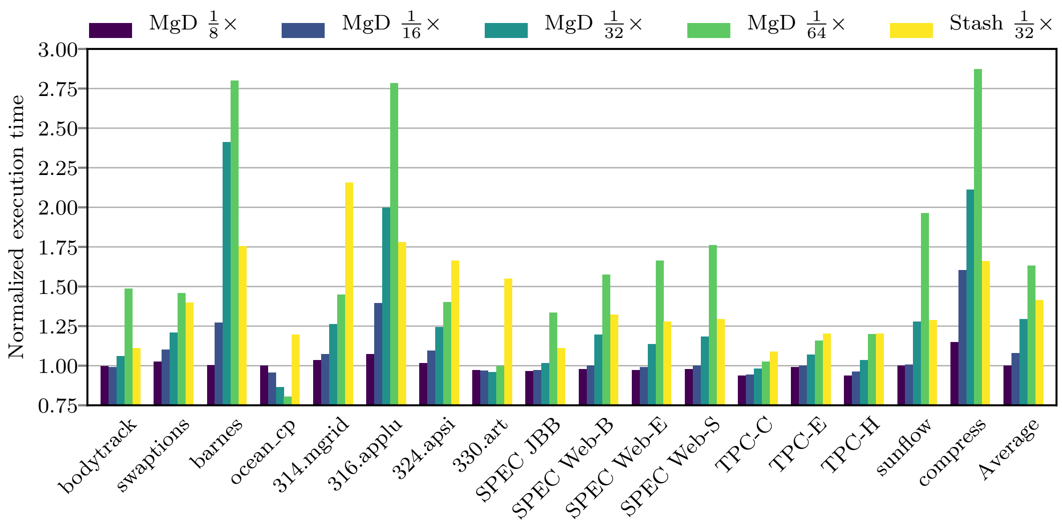

Figure 4.22: Performance of $\frac{1}{8}\times$, $\frac{1}{16}\times$, $\frac{1}{32}\times$, $\frac{1}{64}\times$ multi-grain directory (MgD) and $\frac{1}{32}\times$ Stash directory normalized to a sparse $2\times$ directory.

results on a 128-core system for a wide range applications show that our proposal operating with tiny directories of size ranging between $\frac{1}{32}\times$ and $\frac{1}{256}\times$ performs within 1% of a system with a $2\times$ sparse directory.

Chapter 5

Sharing-aware Private Caching

The emerging many-core server processors with tens of cores are equipped with a per-core private cache hierarchy and a large multi-banked on-die shared last-level cache (LLC). The cores along with their private cache hierarchy and the LLC banks are distributed over a scalable on-die interconnect. Any communication between the cores' private caches and the LLC banks must traverse the interconnect. Due to the traversal through the interconnect, the core cache miss requests can experience large average round-trip latencies even if they hit in the LLC, particularly for the scale of the machine we are considering in this thesis. As the system grows in terms of core-count, the average round-trip LLC hit latency as well as the volume of traffic in the interconnect typically increase making efficient private caching an important requirement for such systems. In this chapter, we squarely focus on the problem of architecting an efficient private cache hierarchy for many-core server processors running multi-threaded workloads drawn from the domains of commercial computing (web serving and data serving) and scientific computing. Traditionally, a two-level private cache hierarchy is used per core where the private L1 and L2 caches treat the private and shared blocks equally. We start our exploration with a baseline design that does not have a private L2 cache allowing us to understand the properties of the L1 cache misses. Our approach is to characterize the cores' L1 cache misses that hit in the LLC and exploit this run-time characterization to eliminate a subset of these misses by architecting a specialized, yet space-efficient, private L2 cache.

To quantify the potential performance improvement achievable by optimizing the core caches, we conduct an experiment on a simulated 128-core server processor with each core having private instruction and data L1 caches (32 KB 8-way each) and a shared 32 MB 16-way LLC partitioned into 128 set-interleaved banks. A 256 KB 16-way LLC bank is attached to a core tile and the 128 tiles are arranged in a 16×8 mesh interconnect exercising dimension-order-routing and having a four-stage routing pipeline at each switch clocked at 2 GHz.[1] In this experiment, the non-compulsory non-coherence L1 cache misses which hit in the LLC are assumed to hit in the L1 cache i.e., they are charged only the L1 cache lookup latency and do not generate any interconnect traffic. We note that the compulsory and coherence misses cannot be reduced in number by optimizing the private cache hierarchy (assuming a fixed block size).

Figure 5.1 shows the percentage core cache misses saved in this experiment partitioned into code and data misses. On average, 78% core cache misses can be saved. For the web and data serving workloads (TPC, SPEC Web, and SPEC JBB), both code and data contribute significantly to the saved misses, while for the remaining applications, the savings primarily arise from data accesses. Figure 5.2 shows the percentage reduction in execution time when this optimization is applied to only code misses and additional reduction when it is applied to both code and data misses. On average, 30% execution time can be optimized away by eliminating the non-compulsory non-coherence code and data L1 cache misses that hit in the LLC. The savings in the execution time range from 14% (SPEC JBB) to 40% (barnes and TPC-C). When only the non-compulsory non-coherence code misses which hit in the LLC are eliminated, the average saving in execution time is 12%. The savings in execution time correlate well with the volume of saved misses shown in Figure 5.1. Any proposal for optimizing these misses must target both code and data because on average, both contribute significantly to the potential improvement.

Motivated by this large potential improvement in performance, we thoroughly characterize the core cache misses that hit in the LLC (Section 5.1). Our characterization study reveals that a small subset of shared code and data blocks contributes to a large fraction of

[1] The simulation environment is similar to the one used for studying Pool and Tiny directories. Further details of the simulation environment used in this chapter can be found in Section 5.3.

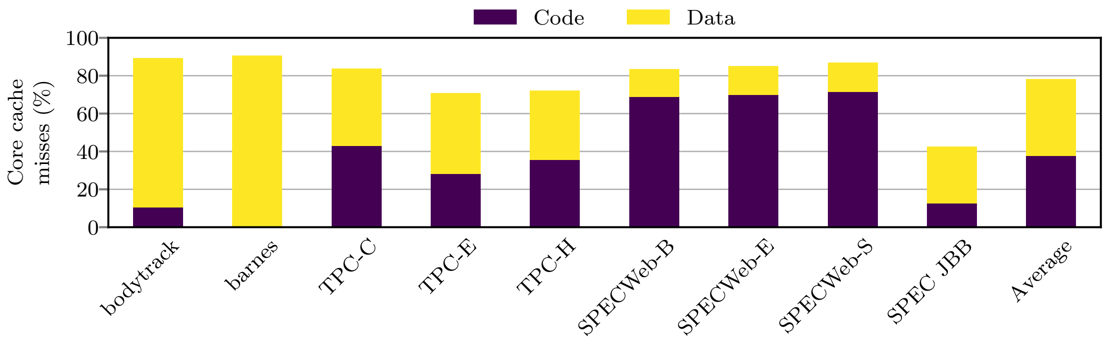

Figure 5.1: Non-compulsory non-coherence core cache misses that hit in LLC.

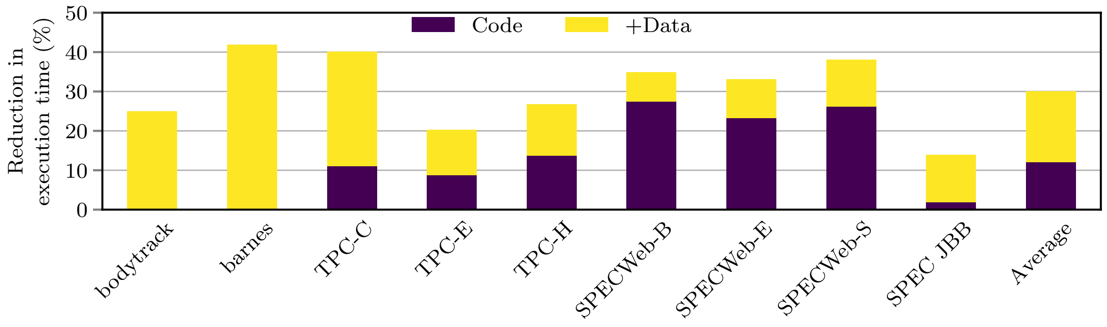

Figure 5.2: Execution time saved when non-compulsory non-coherence core cache misses that hit in the LLC are treated as core cache hits.

the core cache misses that hit in the LLC.[2] This observation leads us to explore the design of a per-core private L2 cache that can serve as an efficient victim cache for the private L1 cache (Section 5.2). The goal of our victim cache designs is to capture the critical subset of the shared blocks. Our best proposal classifies the L1 cache victims into distinct partitions based on two features, namely, an estimate of sharing degree and a simple indirect measure of reuse distance. The collective reuse probability of each partition is learned on-the-fly and used to decide if the L1 cache victims belonging to a partition should be inserted in the victim cache. To the best of our knowledge, this is the first sharing-aware private cache hierarchy design proposal for many-core server processors. Simulation results obtained from a detailed model of a 128-core server processor (Section 5.3) show that our best victim cache design with 64 KB capacity saves 44.1% core cache misses sent to the LLC and 10.6% execution cycles, on average slightly outperforming a traditional

[2] This is in line with what we observed in the last chapter while studying the Tiny directory. There the key finding was that a small fraction of all blocks allocated in the LLC account for a large portion of the LLC accesses to shared blocks. In this chapter, we further generalize it all core cache misses that hit in the LLC.

non-inclusive/non-exclusive per-core 128 KB L2 cache exercising LRU replacement policy (Section 5.4). Further, the savings in core cache misses achieved by our best victim cache proposal are observed to be only 8% less than an optimal victim cache design at 32 KB and 64 KB capacity points.

5.1 Characterization of Core Cache Misses

In this section, we analyze the non-compulsory non-coherence L1 cache misses that hit in the LLC. Since both code and data have important contributions to these misses, this analysis must characterize these misses using features other than code and data. We begin by partitioning these misses based on the sharing types of the LLC blocks being accessed. The LLC block types are discussed below. As the readers will notice, this partitioning is greatly influenced by the way the Tiny directory study partitioned the LLC accesses to shared blocks. An LLC block is said to be temporally private if it never experiences any kind of sharing between more than one core at the same time. A core X accesses such a block from the LLC and caches it privately. It is evicted from the private cache hierarchy of core X before the next LLC access (from the same core X or from a different core Y) to the block. All other LLC blocks are said to be shared. We partition the shared blocks into two groups based on the degree of sharing. We attach a **S**hared **R**ead **A**ccess (SRA) counter with each block to measure its degree of sharing. The SRA counter of a block is initialized to zero when it is filled into the LLC from the main memory. This counter is incremented for a block when an LLC read access (due to a core cache data load or code read miss) to the block hits in the LLC and finds the block in the shared state (S state in MESI coherence protocol). All temporally private blocks have zero SRA. We put all shared blocks with SRA=0 in one group (low degree of sharing) and the remaining shared blocks in another group. A read access to a shared LLC block with zero SRA necessarily finds the block in coherence state M (recall that E and M states are tracked through a single coherence state M in the directory) with the owner being different from the core requesting the read access.

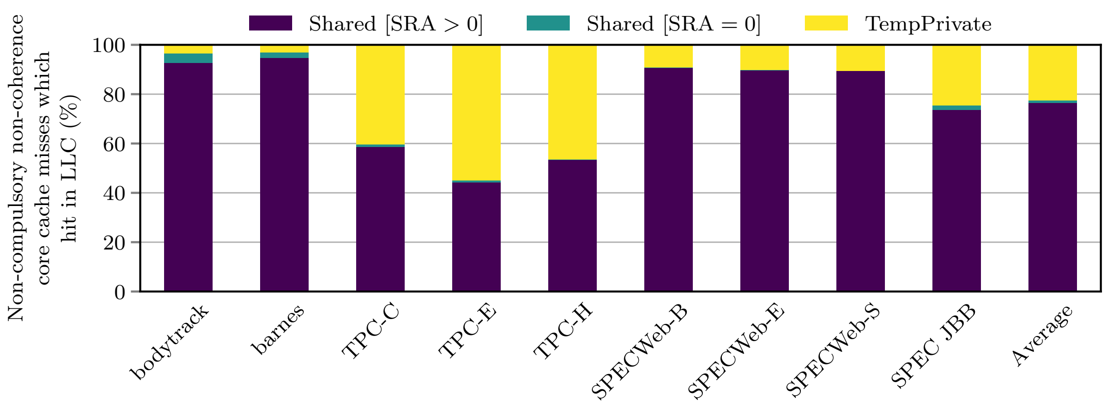

Figure 5.3: Distribution of non-compulsory non-coherence core cache misses that hit in LLC based on the sharing types of the LLC blocks being accessed.

Figure 5.3 shows the distribution of non-compulsory non-coherence L1 cache misses that hit in the LLC. “TempPrivate” represents the temporally private category. On average, 76% of these misses access shared blocks with positive SRA. For all the applications, except TPC-E, more than half of these misses fall in this category. Figure 5.4 shows the percentage reduction in execution time when these misses are treated as L1 cache hits. The bottom segment of each bar shows the percentage reduction in execution time when only the core cache misses to the shared LLC blocks with positive SRA are treated as L1 cache hits. The middle segment of each bar shows the additional saving in execution time when the core cache misses to the shared LLC blocks with zero SRA are also treated as L1 cache hits. The top segment of each bar shows the additional saving in execution time when the core cache misses to the temporally private LLC blocks are also treated as L1 cache hits. On average, 19% execution time can be optimized away by saving the core cache misses which hit the shared LLC blocks with positive SRA. Saving the core cache misses to the shared LLC blocks with zero SRA has negligible impact on performance. These results clearly highlight that saving the core cache misses to the shared LLC blocks with positive SRA is important for performance and interconnect traffic. Figure 5.5 quantifies the percentage of the LLC blocks that are shared and have positive SRAs. On average, just 12% of the LLC blocks fall in this category. Barnes is a clear outlier with 78% of the LLC blocks in this category. Among the rest, only bodytrack and TPC-H have more than 2% of the LLC blocks that are in this category. Therefore, on average, only 12% of the

LLC blocks contribute to 76% of the core cache misses that hit in the LLC and saving these 76% core cache misses can reduce 19% of execution cycles, on average.

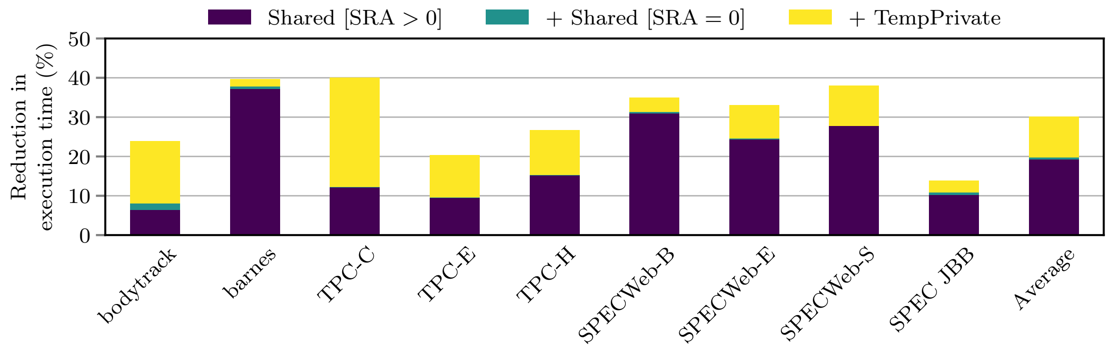

Figure 5.4: Execution time saved when non-compulsory non-coherence core cache misses that hit in the LLC are treated as core cache hits.

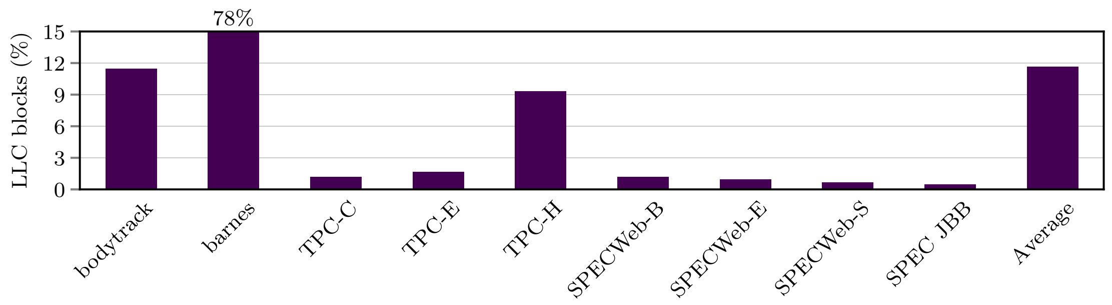

Figure 5.5: Percentage of allocated LLC blocks that are shared with positive SRA.

We further classify the shared LLC blocks with positive SRA based on a normalized SRA ratio. The SRA ratio for an LLC block at any point in time is defined as the ratio of the SRA counter value to the total number of LLC accesses to the block arising from the core cache misses. We classify the shared LLC blocks with positive SRA into three SRA ratio categories, namely C_1, C_2, and C_3. The C_1 category includes all LLC blocks with SRA ratio $\in (0, \frac{1}{2}]$. For C_2 and C_3, the SRA ratio ranges are $(\frac{1}{2}, \frac{3}{4}]$, and $(\frac{3}{4}, 1]$, respectively. Figure 5.6 shows the distribution of the shared LLC blocks with positive SRA. Recall that only 12% of the LLC blocks are shared with positive SRA. Among these, on average, 49%, 10%, and 41% are in C_1, C_2, and C_3, respectively. Figure 5.7 shows the distribution of the non-compulsory non-coherence L1 cache misses that hit in the LLC. On average, 68% of these core cache misses access LLC blocks in C_3 category. This is an important piece of data showing that only 41% of 12% (or, overall 5%) LLC blocks cover 68% of non-compulsory non-coherence L1 cache misses that hit in the LLC. Therefore, it may be possible to capture a significant subset of these L1 cache misses by incorporating a specialized per-core victim cache. Figure 5.8 further shows the percentage reduction in execution time when non-compulsory non-coherence L1 cache misses that hit in the LLC are saved and treated as L1 cache hits. For each application, we show the gradual reduction in execution time as core cache misses to C_3, $C_3+C_2+C_1$, $C_3+C_2+C_1+$ shared with zero SRA, and all LLC blocks are saved. On average, 15.5% execution time can be optimized away by saving the core cache misses to the C_3 blocks. This observation further highlights the fact that a small critical subset of the shared blocks is responsible for majority of the core cache misses, and by saving core cache misses for these blocks significant performance improvement can be achieved.

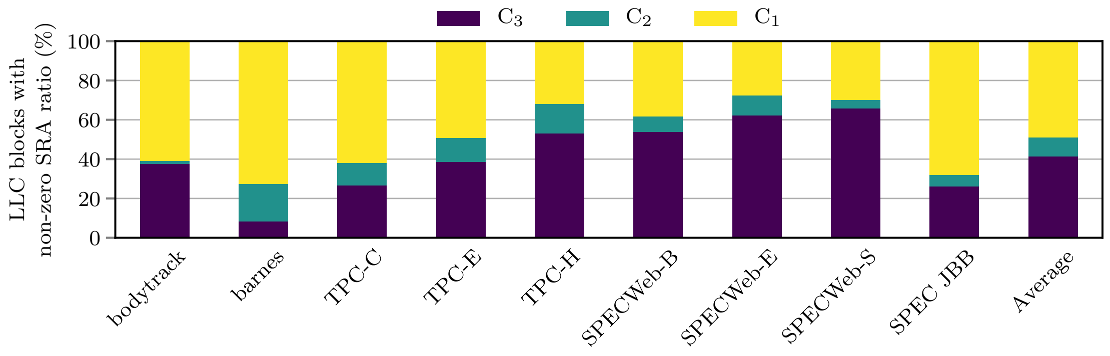

Figure 5.6: Distribution of the shared LLC blocks into the SRA ratio categories.

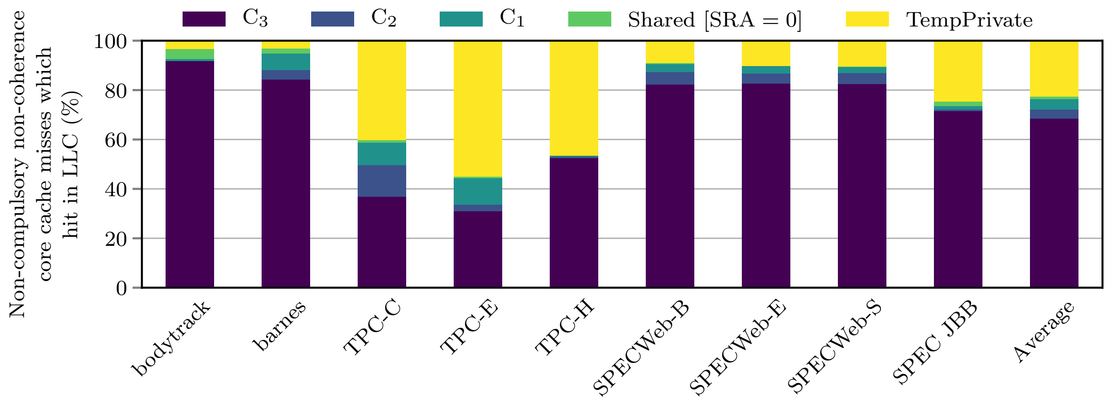

Figure 5.7: Distribution of non-compulsory non-coherence core cache misses that hit in LLC based on the sharing status of the LLC block being accessed.

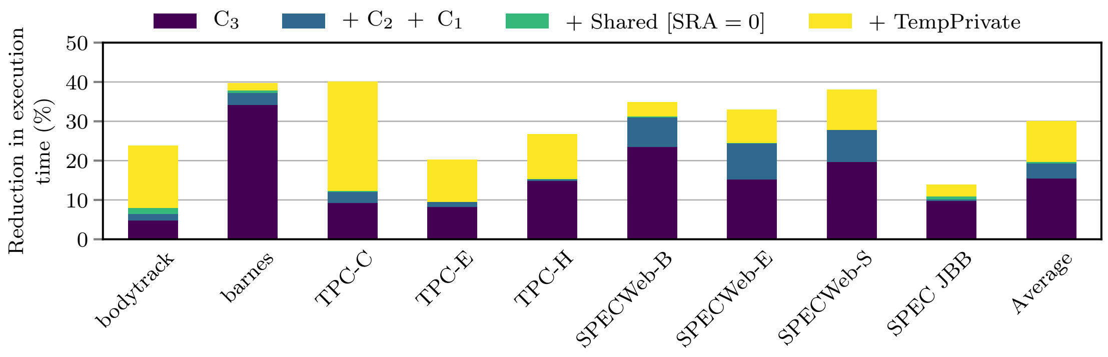

Figure 5.8: Execution time saved when non-compulsory non-coherence core cache misses that hit in the LLC are treated as core cache hits.

5.2 Victim Cache Design

In this section, we architect a private per-core unified victim cache (VC) to capture a subset of the L1 instruction and data cache victims. We begin our discussion by reviewing the basic VC architecture that admits all L1 cache victims (Section 5.2.1). Next, we present two design proposals for selective victim caching (Section 5.2.2) that exploit the findings of our characterization results. All the VC designs considered in this paper are 8-way set-associative. The L1 cache and the VC are looked up serially to avoid lengthening the L1 cache access latency. On an L1 cache miss, the VC is looked up. On a VC hit, the block is invalidated from the VC and copied to the L1 instruction or data cache depending on the request type. On a VC miss, the block is fetched from the outer levels of the memory hierarchy (LLC or main memory) and inserted into the L1 instruction or data cache. As a result, the VC is equivalent to a private per-core L2 cache that is exclusive of the L1 caches.

5.2.1 Victim Caching without Selection

The traditional VC architecture admits all L1 cache victims. We evaluate two replacement policies for such a VC. The first one evicts the least-recently-filled (LRF) block in a VC set.[3] This design requires three replacement state bits per block in an 8-way cache. This design will be referred to as LRF-VC. The second design devotes only one replacement

[3] A least-recently-used replacement policy has no meaning in a VC because on a VC hit, a block is invalidated.

state bit per block. This bit is set to one when a block is inserted into the VC. If all blocks in a set have this bit set to one, all the bits in that set are reset except the bit corresponding to the most recently filled block. Within a set, the replacement policy victimizes the block with the replacement state bit reset; a tie among multiple such eligible candidates is broken by victimizing the block with the lowest physical way id. This replacement policy will be referred to as not-recently-filled (NRF) and this design will be referred to as NRF-VC. The NRF policy is motivated by the observation that the first order locality of a block inserted in the VC is already filtered by the L1 cache and therefore, a precise fill order as maintained by the LRF policy may not be necessary to achieve good performance.

5.2.2 Selective Victim Caching

The two selective victim caching proposals discussed in this section constitute the crux of our contributions. Our L1 cache miss characterization study has established that the selective victim caching proposals must primarily target the shared LLC blocks with positive SRA ratio and that the C_3 blocks are particularly important. In addition to the three categories (C_1, C_2, and C_3) of non-zero SRA ratio, we use C_0 to denote the category of shared as well as temporally private blocks with zero SRA ratio. To identify the category of an LLC block, the SRA ratio needs to be estimated online. For this purpose, two six-bit saturating counters, namely SRA Counter (SRAC) and **O**ther **A**ccess **C**ounter (OAC), are maintained for the block. While this requirement is similar to what we proposed for the Tiny directory design, in the study of this chapter, we use a traditional full-map sparse directory. The SRAC is incremented on LLC read accesses which find the block being requested in the shared state. The OAC is incremented on all other LLC accesses (except writeback) to the block. Both the counters of the block are halved when any of the counters has saturated. The SRA ratio estimate for the block is given by the fraction $\frac{SRAC}{SRAC+OAC}$. The sparse directory entry that tracks coherence of a block is extended by twelve bits to accommodate the two counters. Once a block returns to the unowned/non-shared state, the counters are reset and the SRA ratio for the block is deemed zero. Also, when the sparse directory entry of a block is evicted, its SRA ratio is assumed to become zero. When a block is fetched from the LLC into the L1 cache, the block's SRA ratio category is also fetched and maintained by extending the L1 cache tag by two bits. A block fetched from main memory (due to LLC miss) is assumed to belong to category C_0. When an L1 cache block is evicted, its SRA ratio category is used in deciding whether the block should be admitted into the VC, as discussed in the following designs. If an L1 cache victim is allocated in the VC, its SRA ratio category is also maintained in the VC by extending the VC tag by two bits. When a VC entry is copied into the L1 cache, its SRA ratio category is also copied.

SRA-gNRF-VC

Our first selective VC design tries to capture a subset of the high SRA ratio blocks and implements a generational NRF (gNRF) replacement policy. First, we discuss the gNRF policy, which is inspired by the gNRU policy of the Tiny directory proposal. The design divides the entire execution into intervals or generations. Each VC entry is extended with two state bits, namely, a fill (F) bit and an eviction priority (EP) bit. When a VC block is filled, the F bit of the block is set and the EP bit is reset, recording the fact that the block has been recently filled and must not be prioritized for eviction in the current interval. At the end of each interval, the EP bit of a VC entry is set to the inverse of the F bit signifying that the entry can be considered for eviction in the next interval if and only if the F bit is reset. The F bits of all VC entries are gang-cleared at the beginning of each interval signifying the start of a new generation. Thus, a VC block becomes eligible for eviction within two consecutive intervals.

Now, we discuss the victim caching protocol. On receiving an L1 cache victim block B, the SRA-gNRF-VC design first looks for an invalid way in the target VC set. If there is no such way, it locates the way w with the lowest SRA ratio category (say, C_i) in the target set. If there are multiple ways with the lowest SRA ratio category, the ones with their EP bits set are selected and then among them the one with the lowest physical way id is selected. Let the SRA ratio category of the L1 cache victim block B currently being considered for allocation in the VC be C_j. The SRA-gNRF-VC design victimizes the entry

w (currently holding a block belonging to category C_i) to allocate block B only if one of the following two conditions is met: (i) $i < j$, (ii) $i == j$ and the EP bit of w is set. The first condition helps attract a subset of high SRA ratio blocks into the VC, while the second condition creates an avenue for replacing useless VC entries of a certain SRA ratio category.

We set the generation length to the interval between the fill and a hit to a VC entry, averaged across all entries that experience hits. We dynamically estimate this interval as follows. The interval length is measured in multiples of 4K cycles and the maximum interval length that our hardware can measure is 4M cycles. The VC controller maintains a ten-bit counter T which is incremented by one every 4K cycles (measured using a twelve-bit counter P). Each VC entry is extended by ten bits to record the value of counter T whenever the entry is filled. On a hit to a VC entry, the value of counter T recorded at the time of fill in the entry (T_{fill}) is compared with the current value of counter T ($T_{current}$). If $T_{fill} < T_{current}$, the difference between $T_{current}$ and T_{fill} is added to a counter A maintained in the VC controller. The counter A records the accumulated time between a fill and a hit to a VC entry. Another counter B maintained in the VC controller records the number of values added to counter A. At any point in time, the generation length is estimated as $\frac{A}{B}$. At the beginning of an interval, this value is copied to a generation length counter (GLC), which is decremented by one every 4K cycles. A generation ends when this counter becomes zero. Both the counters A and B are halved when either of them has saturated. When counter T saturates, it is reset to zero.

Overall, fourteen state bits are required per VC entry for implementing the SRA-gNRF policy (SRA ratio category: 2 bits, T_{fill}: 10 bits, and F and EP bits) and two additional bits per L1 cache entry for maintaining the SRA ratio category. For a 64 KB VC and 32 KB instruction and data L1 caches with 64-byte blocks, this overhead is equivalent to 16K bits (2 KB) per core. The counters T, P, A, B, and GLC require a few tens of bits per core.

SRA-VCUB-RProb-VC

The SRA-gNRF-VC design discussed in the last section assumes that the high SRA ratio blocks will enjoy hits in the VC. However, this may not be true in all phases of execution. Additionally, this design also loses opportunity of caching some of the lower SRA ratio blocks that may enjoy some hits in the VC. The SRA-VCUB-RProb-VC design remedies these problems by directly considering the probability that an L1 cache victim would be reused from the VC. It partitions the L1 cache victims into several categories, estimates the collective reuse probability (RProb) of each category, and caches only the L1 cache victims belonging to the categories with high enough reuse probability. Additionally, this design substitutes the gNRF policy by a more efficient replacement policy incorporating four possible ages of a block. This policy, like gNRF, requires only two replacement state bits used to encode four possible ages of a block in the VC.

The L1 cache victims are partitioned online based on the SRA ratio categories and a simple estimate of reuse distance. The reuse distance estimate is obtained as follows. The private cache residency of a block begins when it is fetched into the L1 cache from either LLC or main memory. Its private cache residency ends when it is evicted from the VC or from the L1 cache and not admitted to the VC. During this private cache residency, the block may make multiple trips between the L1 cache and the VC. If a block enjoys at least one use in the VC, it indicates a relatively short reuse distance of the block. We use this indication as an estimate of the reuse distance of the block. This is recorded by maintaining a VC use bit (VCUB) per block in the L1 cache and the VC. When a block is fetched into the L1 cache from the LLC or the main memory, its VCUB is set to zero. The VCUB of a block becomes one when it experiences its first hit in the VC. After this, the VCUB remains set during the rest of the block's private cache residency. An L1 cache victim having VCUB=0 is estimated to have a relatively larger reuse distance and smaller reuse probability compared to an L1 cache victim with VCUB=1. The VCUB induces a top-level partitioning of the L1 cache victims.

The L1 cache victims with VCUB=0 are further partitioned into four classes based on the victims' SRA ratio categories (C_0, C_1, C_2, C_3). For each category, the collective probability of reuse in the VC is estimated online as follows. Eight sets are sampled from the VC and the accesses to these sampled sets for blocks with VCUB=0 are used to estimate the reuse probabilities. The VC controller maintains two counters for each SRA ratio category C_i. One counter (f_i) maintains the number of fills to the sampled sets for category C_i blocks with VCUB=0. The other counter (h_i) maintains the number of hits in the sampled sets experienced by the blocks with VCUB=0 and category C_i. The reuse probability p_i of category C_i given VCUB=0 is h_i/f_i. Periodically, all the eight counters are halved.

Next, we discuss the victim caching protocol of the SRA-VCUB-RProb-VC design. The following two principles guide the VC allocation policy. First, all L1 cache victims mapping to the sampled sets are allocated in the VC because the reuse probabilities are learned from the behavior of the blocks in the sampled sets. Second, the age assigned to a block allocated in the VC is zero, two, or three depending on the estimated reuse probability of the partition containing the block. A higher reuse probability is associated with a lower age, which, in turn, signifies a lower eviction priority. The L1 cache victims with VCUB=1 are assumed to have the maximum reuse probability. The dynamic reuse probability of the L1 cache victims with VCUB=0 are estimated on-the-fly, as already discussed. On receiving an L1 cache victim block B, if the VCUB of B is set, it is allocated in the VC and assigned an age zero. If the VCUB of B is reset and B maps to a sampled set, it is allocated in the VC and assigned an age two. If the VCUB of B is reset and B does not map to a sampled set, its SRA ratio category C_i decides further actions. Let the current reuse probability estimate of C_i be p_i. If p_i is at least 1/8 (implemented as $h_i \geq [f_i$ shifted right by 3 bit positions$]$), the block B is allocated in the VC and assigned an age two. If p_i is less than 1/8, but there is an invalid way in the target VC set, the block B is allocated in that way and assigned an age three. If p_i is less than 1/8 and there is no inavlid way, the block B is not allocated in the VC. We have experimented with four reuse probability thresholds, namely, 1/2, 1/4, 1/8, and 1/16. Among these, 1/8 is found to achieve the best performance. Tables 5.1 and 5.2 summarize the VC operations.

Within a set, the VC policy evicts a block with age three; a tie among multiple such

Table 5.1: VC allocation protocol for an L1 cache victim block

Block attributes	Maps to a VC sample set	Doesn't map to a VC sample set
VCUB=1	Allocate with age=0	Allocate with age=0
VCUB=0; SRA category C_i; $p_i \geq 1/8$	Allocate with age=2; f_i++	Allocate with age=2
VCUB=0; SRA category C_i; $p_i < 1/8$		Allocate with age=3 if an invalid way is available

Table 5.2: VC actions on a hit

Block attributes	Maps to a VC sample set	Doesn't map to a VC sample set
VCUB=1	Invalidate; copy to L1	Invalidate; copy to L1
VCUB=0; SRA category C_i	Invalidate; copy to L1; h_i++; VCUB$\leftarrow$1	Invalidate; copy to L1; VCUB$\leftarrow$1

blocks is broken by victimizing the block at the lowest physical way. If no such block exists in the set, the ages of all blocks in the set are incremented until a block with age three is found. This replacement logic is similar to the static re-reference interval prediction (SRRIP) policy [41]. Our policy for inserting a block into the VC as discussed above is, however, entirely different from what SRRIP uses as its insertion policy (inserts always at age two).

Overall, five state bits are required per VC entry for implementing the SRA-VCUB-RProb policy (SRA ratio category: 2 bits, VCUB: 1 bit, age: 2 bits) and three extra bits per L1 cache entry (SRA ratio category: 2 bits, VCUB: 1 bit). For a 64 KB VC and 32 KB instruction and data L1 caches with 64-byte blocks, this overhead is equivalent to 8K bits (1 KB) per core. The additional overhead of the h_i and f_i counters (nine bits each) is 72 bits per core.

5.3 Simulation Environment

The simulation environment is similar to the one used for Tiny directory study. We use an in-house modified version of the Multi2Sim simulator [78] to model a chip-multiprocessor having 128 dynamically scheduled out-of-order issue x86 cores clocked at 2 GHz. The details of the baseline configuration are presented in Table 5.3. The interconnect switch microarchitecture assumes a four-stage routing pipeline with one cycle per stage at 2 GHz clock. The stages are buffer write/route computation, virtual channel allocation, output port allocation, and traversal through switch crossbar. There is an additional 1 ns link latency to copy a flit from one switch to the next. The overall hop latency is 3 ns.

We evaluate our VC proposals for two configurations, namely, 32 KB 8-way and 64 KB 8-way. These have lookup latencies of one and two cycles, respectively. We also explore how our proposals fare against traditional non-inclusive/non-exclusive 8-way L2 caches of capacity 32 KB, 64 KB, and 128 KB exercising fill-on-miss and LRU as well as state-of-the-art replacement policies. These L2 cache configurations have lookup latencies of one, two, and three cycles, respectively. The latencies have been fixed using CACTI [37] assuming 22 nm technology node (we use the version of CACTI distributed with McPAT [38]).

Table 5.3: Baseline simulation environment

On-die cache hierarchy, interconnect, and coherence directory
Per-core iL1 and dL1 caches: 32 KB, 8-way, LRU, latency 1 cycle
Shared LLC: 32 MB, 16-way, 128 banks, LRU, bank lookup latency 4 cycles for tag + 2 cycles for data, non-inclusive/non-exclusive, fill on miss, no back-inval. on eviction
Cache block size at all cache levels: 64 bytes
Interconnect: 2D mesh clocked at 2 GHz, two-cycle link latency (1 ns), four-cycle pipelined routing per switch (2 ns latency); Routing algorithm: dimension-order-routing; Each switch connects to: a core, its L1 caches, one LLC bank, one 4$\times$ (relative to per-core L1 caches) sparse directory slice [35, 66].
Sparse directory slice: 16-way, LRU replacement
Coherence protocol: write-invalidate MESI
Main memory
Memory controllers: eight single-channel DDR3-2133 controllers, evenly distributed over the mesh, FR-FCFS scheduler
DRAM modules: modeled using DRAMSim2 [68], 12-12-12, BL=8, 64-bit channels, one rank/channel, 8 banks/rank, 1 KB row/bank/device, x8 devices, open-page policy

The applications for this study are drawn from various sources and detailed in Table 5.4 (ROI refers to the parallel region of interest). Since many-core shared memory

Table 5.4: Simulated applications

Suite	Applications	Input/Configuration	Simulation length
PARSEC	bodytrack	sim-medium	Complete ROI
SPLASH-2[4]	barnes	32K particles	Complete ROI
SPEC JBB	SPEC JBB	82 warehouses, single JVM instance	Six billion instructions
TPC	MySQL TPC-C	10 GB database, 2 GB buffer pool, 100 warehouses, 100 clients	500 transactions
	MySQL TPC-E	10 GB database, 2 GB buffer pool, 100 clients	Five billion instructions
	MySQL TPC-H	2 GB database, 1 GB buffer pool, 100 clients, zero think time, even distribution of Q6, Q8, Q11 Q13, Q16, Q20 across client threads	Five billion instructions
SPEC Web	Apache HTTP server v2.2 running Banking, Ecommerce, Support	Worker thread model, 128 simultaneous sessions, mod_php module	Five billion instructions

[4] The SPLASH-2 applications are drawn from the SPLASH2X extension of the PARSEC distribution.

server processors are prevalently used for commercial computing, we pick seven of our nine applications used in this study from the domain of web and data serving (SPECWeb-B, SPECWeb-E, SPECWeb-S, TPC-C, TPC-E, TPC-H, SPEC JBB). Additionally, we pick one application (barnes) as a representative of scientific computing, which often exercise large-scale shared memory servers. One application (bodytrack) is selected from the domain of computer vision where parallel processing is quite popular. The inputs, configurations, and simulation lengths are chosen to keep the simulation time within reasonable limits while maintaining fidelity of the simulation results. The PARSEC and SPLASH-2 applications are simulated in execution-driven mode, while the rest of the applications are simulated by replaying an instruction trace collected through the PIN tool capturing all activities taking place in the application address space. The PIN trace is collected on a 24-core machine by running each multi-threaded application creating at most 128 threads (including server, application, and JVM threads). Before replaying the trace through the simulated 128-core system, it is pre-processed to expose maximum possible concurrency across the threads while preserving the global order at global synchronization boundaries and between load-store pairs touching the same 64-byte block.

5.4 Simulation Results

In this section, we present a detailed evaluation of our proposal. Section 5.4.1 analyzes the performance of our proposal, while Section 5.4.2 examines the interconnect traffic of the memory system of our proposal. Section 5.4.3 further quantifies the energy expended in the cache hierarchy by our proposal. All results are normalized to a baseline design with 32 KB 8-way instruction and data L1 caches per core and no L2 cache. The shared LLC is 32 MB 16-way in all configurations.

5.4.1 Performance Evaluation

We begin the discussion on performance evaluation by comparing the four VC designs presented in Section 5.2. Figure 5.9 quantifies the percentage reduction in core cache misses relative to the baseline for the four VC designs, namely LRF-VC, NRF-VC, SRA-gNRF-VC, and SRA-VCUB-RProb-VC. We have also included the results for an optimal VC design that implements Belady's optimal replacement algorithm [11, 63] extended with the option of not allocating a block in the VC if its next-use distance is larger than all blocks in the target set. The optimal design requires knowledge about the future accesses. It is evaluated offline after collecting the access trace for each application. All VC designs have 64 KB 8-way configuration. On average, both LRF and NRF reduce the core cache misses by 40%, while SRA-gNRF achieves a 41.3% reduction. The SRA-VCUB-RProb design achieves a reduction of 44.1% having a less than 8% gap to the optimal design, which achieves a reduction of 51.9%. Compared to LRF and NRF, the top gainers of the SRA-VCUB-RProb design include bodytrack, barnes, and TPC-C. For TPC-H, the SRA-VCUB-RProb design achieves near-optimal core cache misses.

Figure 5.10 presents the percentage reduction in execution time for the four VC designs with 64 KB capacity relative to the baseline. The performance of the optimal design cannot be evaluated because the future accesses cannot be fixed online. On average, LRF and NRF save 9% execution time, while the SRA-gNRF and SRA-VCUB-RProb designs reduce execution time by 9.9% and 10.6%, respectively. Within each application, the savings in

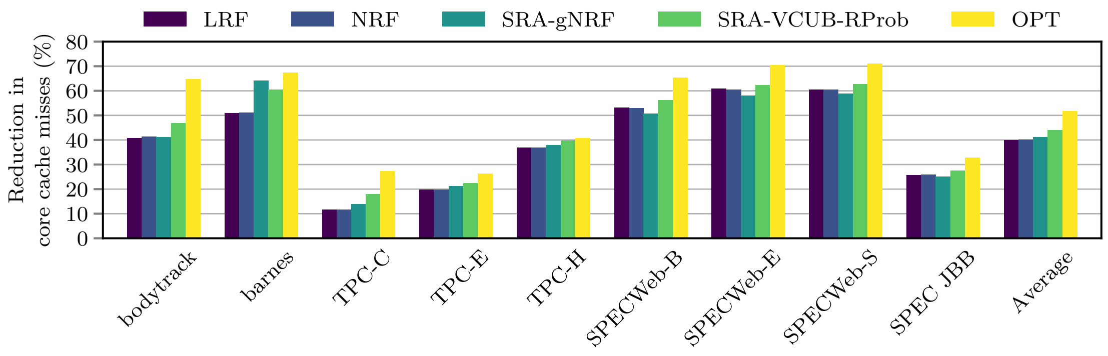

Figure 5.9: Reduction in core cache misses with 64 KB VC relative to baseline.

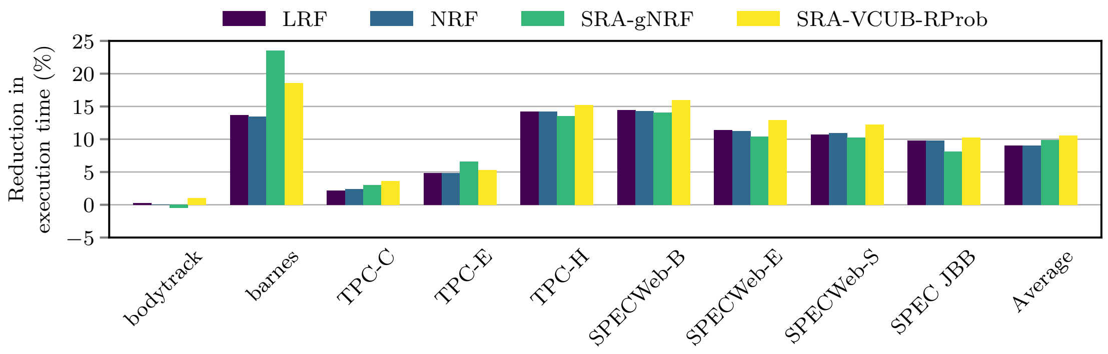

Figure 5.10: Reduction in execution time with 64 KB VC relative to baseline.

execution time correspond well to the relative trend shown in Figure 5.9. Bodytrack fails to improve much in performance because saving core cache misses is not particularly important for its performance.

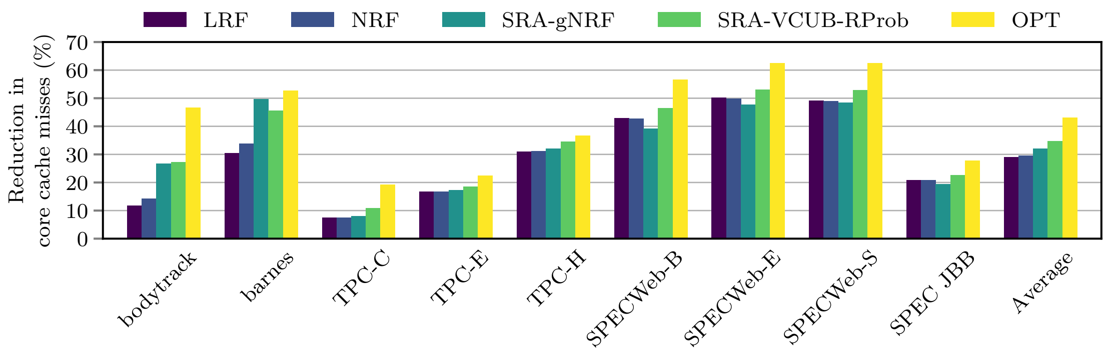

Figure 5.11: Reduction in core cache misses with 32 KB VC relative to baseline.

Figures 5.11 and 5.12 evaluate the VC designs with 32 KB capacity. All designs continue to be 8-way set-associative. On average, the SRA-VCUB-RProb design saves 35% core cache misses relative to the baseline and is only 8% away from the optimal design,

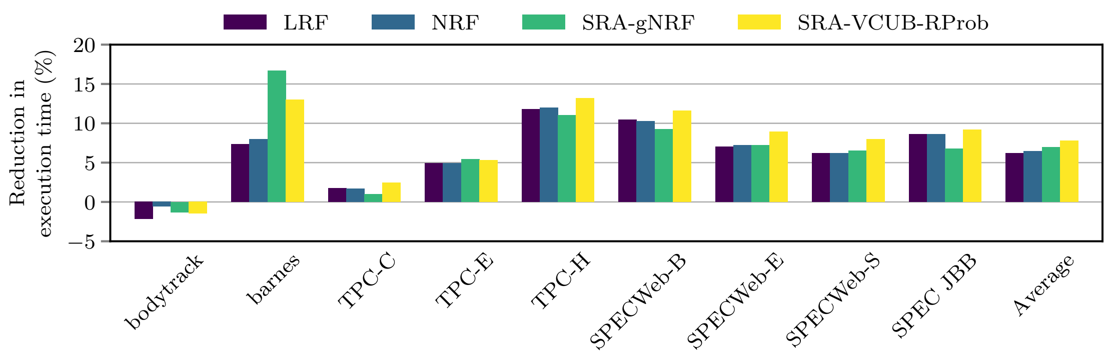

Figure 5.12: Reduction in execution time with 32 KB VC relative to baseline.

which saves 43% core cache misses. Compared to LRF and NRF, bodytrack and barnes continue to enjoy large savings in core cache misses with the SRA-VCUB-RProb design. The SRA-VCUB-RProb design achieves a 7.9% reduction in execution time compared to the baseline, on average. We will consider only the best performing VC design i.e., SRA-VCUB-RProb in further evaluation.

Next, we compare the SRA-VCUB-RProb design with the traditional non-inclusive/non-exclusive L2 caches exercising LRU replacement policy. Figure 5.13 shows the percentage reduction in core cache misses relative to the baseline for 32 KB and 64 KB SRA-VCUB-RProb-VC design and 32 KB, 64 KB, and 128 KB traditional non-inclusive/non-exclusive L2 cache design. On average, the 32 KB traditional L2 cache design saves only 6% core cache misses, while the 32 KB VC design saves an impressive 35% core cache misses; the 64 KB traditional L2 cache design saves 22.3% core cache misses, while the 64 KB VC design saves 44.1% core cache misses. Most importantly, a 128 KB traditional L2 cache design saves 42.2% core cache misses. This saving is a couple of percentages *lower* than what a half-sized (64 KB) VC exercising the SRA-VCUB-RProb design achieves. Similarly, a 32 KB VC saves much higher percentage of core cache misses than a traditional 64 KB L2 cache. Compared to 128 KB traditional L2 cache, our 64 KB SRA-VCUB-RProb-VC design saves 7.78 MB of on-die SRAM storage for our 128-core configuration assuming 48-bit physical address (our proposal's overheads of VC and L1 cache state bits and additional 12 bits per directory entry are accounted for).

Figure 5.14 compares the SRA-VCUB-RProb-VC design with the traditional L2 caches in terms of percentage saving in execution time relative to the baseline. Across the board, the VC design outperforms the same-sized traditional L2 caches by large margins. More importantly, a 32 KB VC outperforms a 64 KB traditional L2 cache and a 64 KB VC outperforms a 128 KB traditional L2 cache, on average. These results strongly advocate the replacement of traditional private L2 caches by specialized per-core VC designs that can achieve significant space saving per core while delivering better performance at lower interconnect traffic.

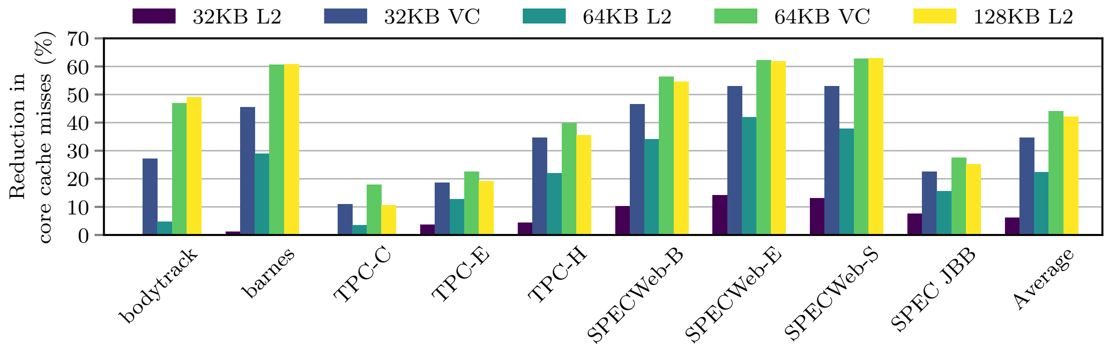

Figure 5.13: Comparison between traditional non-inclusive L2 cache and SRA-VCUB-RProb-VC in terms of reduction in core cache misses.

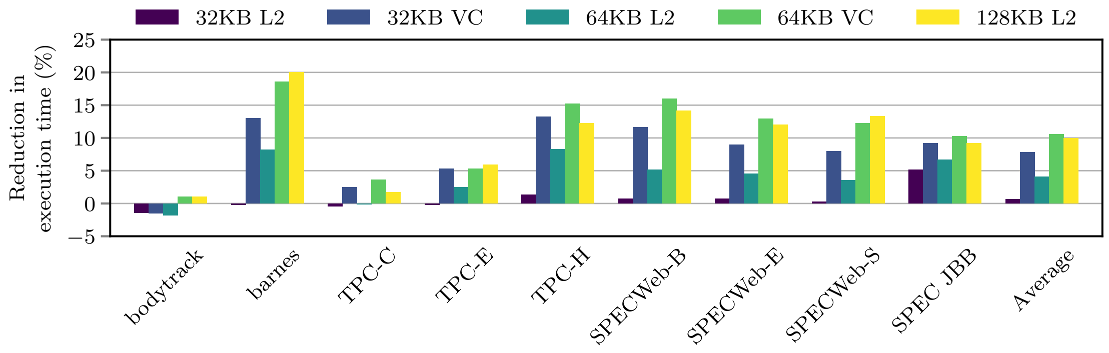

Figure 5.14: Comparison between traditional non-inclusive L2 cache and SRA-VCUB-RProb-VC in terms of reduction in execution time.

Figures 5.15 and 5.16 compare our SRA-VCUB-RProb-VC design with iso-capacity baselines. A configuration with a 32 KB VC invests a total of 96 KB cache per core when the 32 KB L1 instruction and data cache capacities are included. This configuration should be compared against a baseline that also invests 96 KB of L1 caches per core. So, we evaluate a configuration that has 48 KB 6-way L1 instruction and data caches per core.

Similarly, we compare the 64 KB VC design with a configuration that has 64 KB 8-way L1 instruction and data caches per core. The 48 KB L1 caches have a single-cycle lookup latency, while the 64 KB L1 caches have two-cycle lookup latency. The results are shown relative to the baseline with 32 KB L1 instruction and data caches per core. Figures 5.15 and 5.16 respectively show that the 48 KB L1 cache configuration saves 20% core cache misses and 4.6% execution time, while the 32 KB VC saves 35% core cache misses and 7.9% execution time, on average. Both 32 KB and 64 KB VC designs save more core cache misses and execution time than the 64 KB L1 cache configuration, on average. The 64 KB L1 cache configuration saves 32.5% core cache misses and 6.3% execution time, while the 64 KB VC saves 44.1% core cache misses and 10.6% execution time, on average. Overall, for all applications, the SRA-VCUB-RProb-VC design comfortably outperforms the iso-capacity baseline configurations.

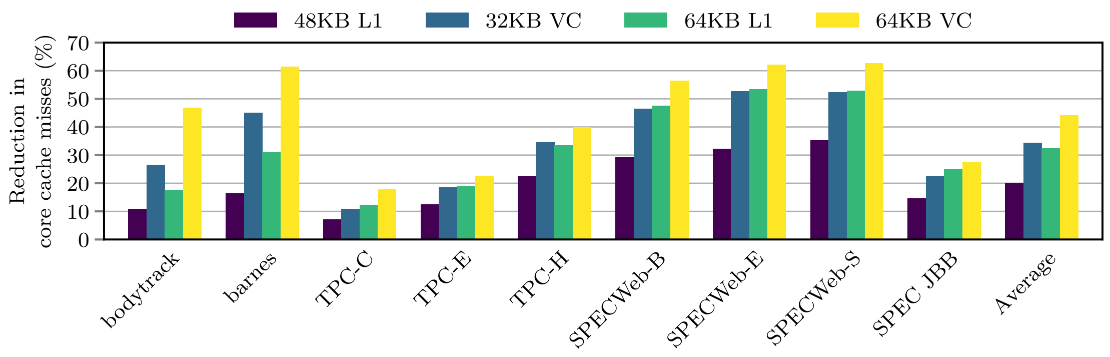

Figure 5.15: Comparison between iso-capacity L1-only baselines and the SRA-VCUB-RProb-VC configurations in terms of reduction in core cache misses.

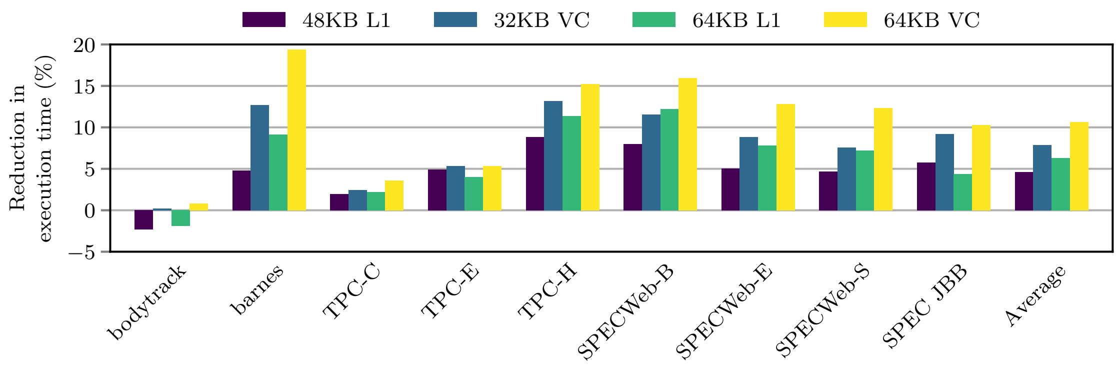

Figure 5.16: Comparison between iso-capacity L1-only baselines and the SRA-VCUB-RProb-VC configurations in terms of reduction in execution time.

A cost-effective alternative to designing a specialized VC is to enhance the L1 caches in a traditional single-level private cache hierarchy and the L2 cache in a traditional two-level private cache hierarchy with state-of-the-art replacement/insertion policies. We explore how the configurations equipped with the SRA-VCUB-RProb-VC design fare against the traditional single-level (L1 cache only) and two-level (L1 and L2 caches) private cache hierarchies enhanced with the signature-based hit prediction (SHiP) policy proposal [84]. The instruction L1 and the data L1 caches of the traditional single-level private cache hierarchy are enhanced with the SHiP-mem and SHiP-PC policies, respectively. The L2 cache in the traditional two-level private cache hierarchy is enhanced with SHiP-mem for instruction blocks and SHiP-PC for data blocks. We compare these configurations against the SRA-VCUB-RProb-VC design working with the traditional baseline 32 KB L1 caches exercising the LRU replacement policy. Figures 5.17 and 5.18 show the results normalized to the baseline with 32 KB L1 instruction and data caches per core exercising the LRU replacement policy. The single-level private cache hierarchy configurations with L1 caches exercising SHiP policy are shown as 32 KB L1, 48 KB L1, and 64 KB L1. The two-level private cache hierarchy configurations with non-inclusive/non-exclusive L2 cache exercising SHiP policy are shown as 32 KB L2, 64 KB L2, and 128 KB L2. The 32 KB and 64 KB VC configurations use the SRA-VCUB-RProb design along with 32 KB instruction and data L1 caches exercising LRU policy.

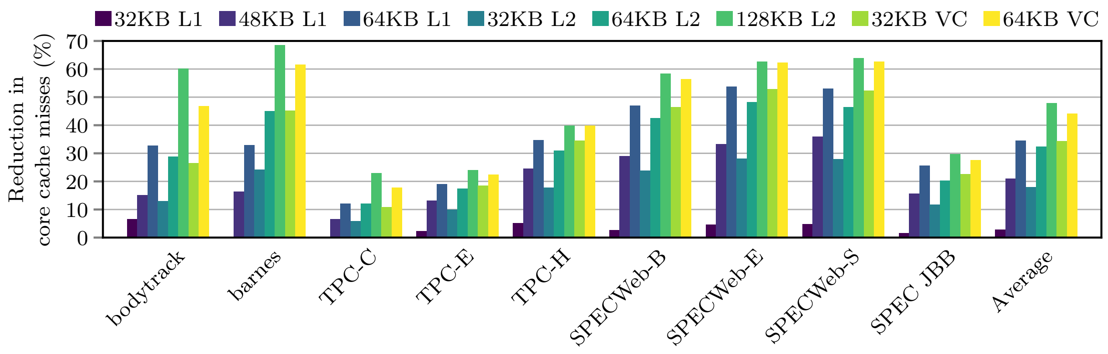

Figure 5.17: Comparison of core cache miss savings between the SRA-VCUB-RProb-VC design and single-level and two-level private hierarchy configurations enhanced with the SHiP policy.

A comparison of Figures 5.17 and 5.18 with Figures 5.15 and 5.16 shows that the SHiP

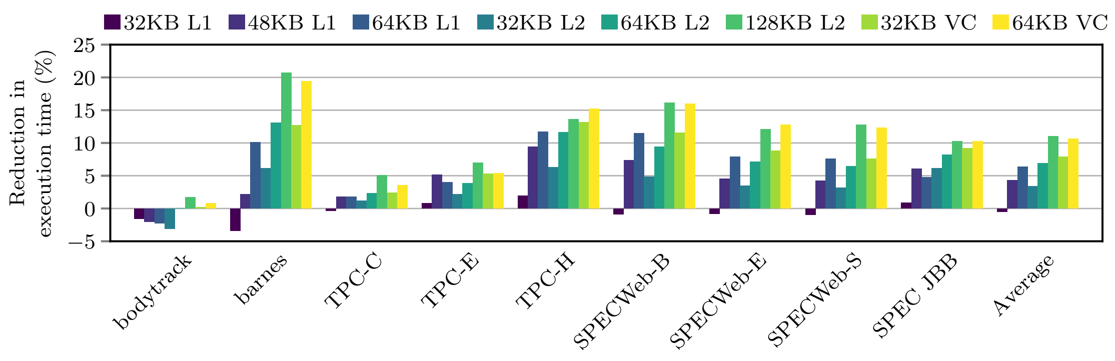

Figure 5.18: Comparison of execution time savings between the SRA-VCUB-RProb-VC design and single-level and two-level private hierarchy configurations enhanced with the SHiP policy.

policy is not particularly effective for L1 caches and performs close to the LRU policy. On the other hand, a comparison with Figures 5.13 and 5.14 shows that the SHiP policy when implemented in the L2 cache is able to improve the performance by a reasonable amount relative to LRU policy, on average. The SHiP policies are designed to work well for caches that experience an access stream with filtered locality e.g., L2 and outer level caches. Nonetheless, Figures 5.17 and 5.18 show that both 32 KB and 64 KB SRA-VCUB-RProb-VC designs save more core cache misses and execution time than all traditional configurations enhanced with the SHiP policies except the 128 KB L2 cache configuration, on average. The 128 KB L2 cache working with the SHiP policies outperforms the 64 KB VC design only marginally (11.0% vs. 10.6% execution time saving) and saves only a few percentage extra core cache misses (47.8% vs. 44.1%), on average. Overall, the SRA-VCUB-RProb-VC design continues to outperform the iso-capacity single-level as well as two-level private cache hierarchies enhanced with the SHiP policies.

In summary, our detailed performance evaluation presents a compelling case for the SRA-VCUB-RProb-VC design as the private per-core L2 cache in many-core server processors. It comfortably outperforms the iso-capacity traditional single-level and two-level non-inclusive/non-exclusive private cache hierarchy designs exercising LRU as well as SHiP policies. Further, a 32 KB VC outperforms a 64 KB traditional L2 cache and a 64 KB VC outperforms a 128 KB traditional L2 cache exercising the LRU policy presenting an opportunity to halve the L2 cache space per core with better performance. When the traditional L2 cache is enhanced with the SHiP policy, the performance of our 64 KB SRA-VCUB-RProb-VC design comes very close to a 128 KB traditional L2 cache.

5.4.2 Interconnect Traffic Comparison

Figure 5.19 compares the on-die interconnect traffic (total number of bytes transferred) for five different private cache configurations. For each application, the leftmost two bars represent single-level private cache configurations with 32 KB and 64 KB L1 caches exercising LRU policy and no L2 cache. The next two bars represent configurations with 64 KB and 128 KB non-inclusive/non-exclusive L2 cache exercising LRU policy. The rightmost bar represents a configuration with 64 KB VC exercising our SRA-VCUB-RProb policy. The last three configurations exercise 32 KB L1 caches with LRU policy. The interconnect traffic for each configuration is divided into four categories. The private cache misses and their responses constitute the processor traffic. The private cache evictions to the LLC and their acknowledgements constitute the writeback traffic. The requests and replies to and from the memory controllers constitute the DRAM traffic. Everything else constitutes the coherence traffic. The results are normalized to the total interconnect traffic of the leftmost bar in each application. Across the board, our VC proposal saves significant portions of the interconnect traffic arising primarily from the savings in processor misses and private cache evictions. Compared to the baseline 32 KB L1 configuration (leftmost bar), our proposal saves 43.1% interconnect traffic. Compared to the 64 KB L1 and 64 KB L2 configurations, our proposal saves 11.5% and 20.9% interconnect traffic, respectively. These two configurations invest the same total cache (128 KB) to the per-core private cache hierarchy as our proposal does using the 64 KB VC. It is encouraging to note that compared to the 128 KB L2 cache configuration, our proposal saves 1.9% interconnect traffic while requiring only half the L2 cache space.

5.4.3 Iso-capacity Energy Comparison

Figure 5.20 quantifies the total energy (dynamic and leakage together) expended by the on-die cache hierarchy and the sparse directory for three different iso-capacity (128 KB per

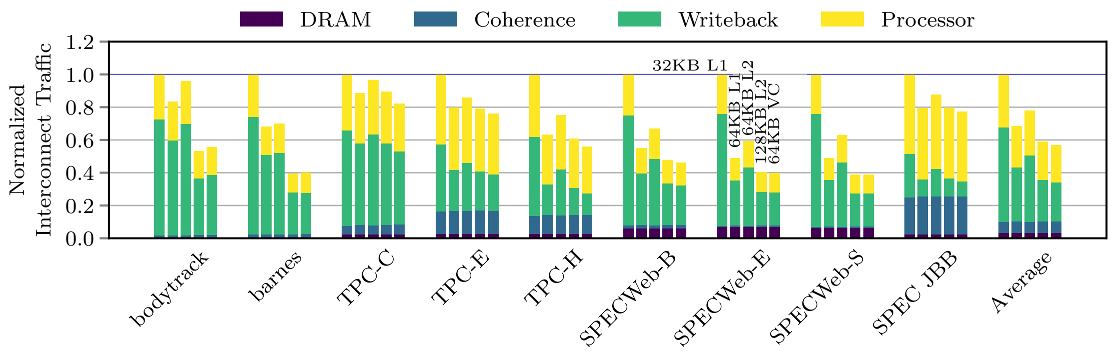

Figure 5.19: Interconnect traffic for private cache hierarchy configurations.

core) private cache hierarchy configurations assuming 22 nm technology nodes (determined with CACTI). For each application, the leftmost bar (marked L1) shows the total energy when the per-core private cache hierarchy has 64 KB 8-way L1 instruction and data caches and no L2 cache. The middle bar (marked VC) shows the total energy when the per-core private cache hierarchy has 32 KB 8-way L1 instruction and data caches and a 64 KB 8-way VC exercising the SRA-VCUB-RProb design. The rightmost bar (marked L2) shows the total energy when the per-core private cache hierarchy has 32 KB 8-way L1 instruction and data caches and a 64 KB 8-way traditional non-inclusive/non-exclusive L2 cache exercising LRU replacement policy. All L1 caches exercise LRU replacement policy. Each bar shows the energy contributed by the instruction L1 cache (IL1), data L1 cache (DL1), L2 cache (L2), LLC, and the coherence directory. All results are normalized to the total energy of the leftmost bar for each application. On average, the VC configuration expends 1% less energy compared to the L1 configuration, while the L2 configuration expends 3% more energy than the L1 configuration. As a result, the 64 KB VC configuration has 6% and 10% less energy-delay product compared to the iso-capacity L1 and L2 configurations, respectively.

5.5 Related Work

Fully-associative very small (8 to 32 entries) victim caches were introduced to handle conflict misses in small direct-mapped caches [44]. The existing selective victim caching proposals explore run-time strategies for identifying the L1 cache evictions resulting from

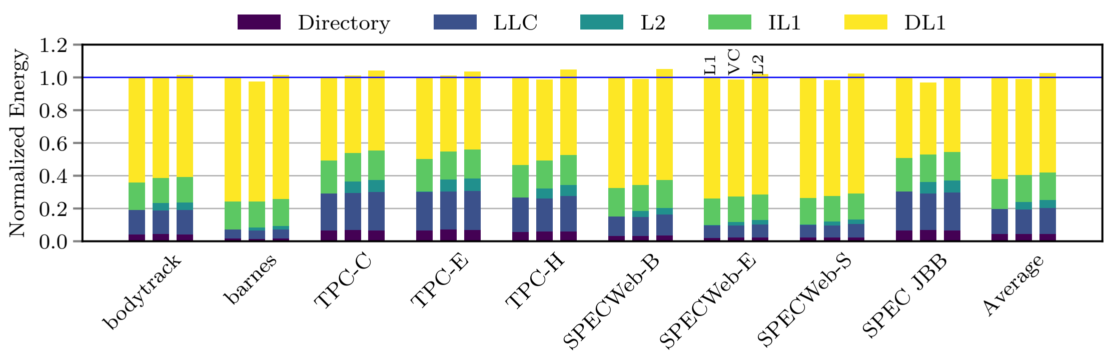

Figure 5.20: Total energy expended by the cache hierarchy for various private cache hierarchy configurations with a per core private cache hierarchy budget of 128 KB.

conflicts and capture these evictions in the fully-associative L1 victim cache of single-core processors [23, 39]. In contrast, our proposal targets larger set-associative victim caches working with L1 caches having high set-associativity where only conflict miss selection is not enough. To the best of our knowledge, this is the first proposal on sharing-aware per-core victim caching for server workloads.

The design of a large victim cache with selective insertion based on frequency of misses has been explored and is shown to work well with large inclusive LLCs [9]. Another proposal exploits the dead blocks in a large inclusive LLC to configure an “embedded” victim cache [50]. These designs are not suitable for per-core mid-level victim caches.

Our victim cache, by design, introduces an exclusive L2 cache in the per-core private cache hierarchy of the many-core processor. The advantages and disadvantages of exclusive LLCs compared to their inclusive counterparts have been explored by several studies [45, 91]. Bypass and insertion policy optimizations for large exclusive LLCs have also been studied [17, 32]. In contrast, we architect a mid-level exclusive victim cache per core in a many-core server processor.

The use of large private caches working with an exclusive LLC has been explored for a small-scale (16 cores) server processor [40]. Our proposal leaves the LLC unchanged and architects a space-efficient private cache hierarchy. Also, specialized coherence protocols that allow selective private caching of certain data based on temporal and spatial locality estimates have been proposed [54]. Our proposal uses a traditional MESI coherence protocol obviating the need to verify a new coherence protocol and designs a small private victim cache per core to optimize the interconnect traffic.

Several studies have explored specialized architectures for the on-die interconnect with the goal of optimizing the latency observed and energy expended in ferrying information between the private cache hierarchy and the shared last-level cache in server processors [51, 58, 59, 60, 79]. The design innovations in these studies involve the following: predicting the useful words within a requested cache block and transmitting only the flits that contain these words [51]; architecting specialized topologies and switches to accelerate instruction delivery to the cores' instruction caches [58]; designing a hybrid of virtual cut-through and circuit switched routing protocols to improve the communication latency [59]; selectively eliminating the per-hop resource allocation delay through proactive resource allocation policies [60]; designing separate request and response networks to suit the different demands of request and response packets [79]. In contrast, our proposal uses a traditional interconnection network and optimizes the interconnect traffic by designing a specialized private cache hierarchy.

A significant body of research has recognized the importance of optimizing the instruction cache performance of the cores in the context of commercial server workloads. One set of studies observes that there is a large amount of overlap within and across transactions in terms of instruction footprint and database operations in online transaction processing (OLTP) workloads. These studies exploit this instruction locality by judicious scheduling and migration of transaction threads and database actions [6, 7, 8, 77]. This class of proposals includes time-multiplexed scheduling of similar threads on a core [6], spreading the overall instruction footprint across the cores and migrating similar threads to a core [7, 8], and scheduling similar database actions on the same core [77]. These techniques require careful synchronization between segments of different transactions and properly scheduling them either temporally or spatially. Another class of proposals has designed specialized instruction prefetchers [28, 30, 46, 47, 48, 52, 53]. While prefetchers can hide the inefficiencies of the private cache hierarchy, they cannot save interconnect traffic. Our proposal, instead, focuses on the design of the private cache hierarchy to improve the interconnect traffic as well as performance. The application domain of our proposal is not limited to just OLTP or instruction delivery for commercial workloads.

Finally, our proposal on ultra-low-overhead coherence tracking has exploited the observation that a small fraction of LLC blocks experience frequent read-sharing [72]. In this chapter, we show that a small fraction of the shared blocks contribute to a large fraction of core cache misses. Our victim cache design captures a subset of these critical blocks.

5.6 Summary

We have presented the designs of a victim cache working with the per-core private L1 caches of a many-core server processor. The victim cache effectively replaces the traditional private L2 cache. Our best victim caching proposal partitions the L1 cache victim space into different classes based on the degree of sharing and an indirect estimate of the reuse distance. It estimates the reuse probability of each partition dynamically and uses these estimates to decide the partitions that should be retained in the victim cache. This selective victim caching proposal, on average, saves 44.1% core cache misses and 10.6% execution time compared to a baseline that does not have a private L2 cache. Further, this proposal comfortably outperforms iso-capacity traditional single-level and two-level private cache hierarchy designs. Most importantly, 32 KB and 64 KB victim caches outperform traditional non-inclusive/non-exclusive 64 KB and 128 KB LRU L2 caches, respectively. This opens up the opportunity of halving the L2 cache space investment per core while offering better performance.

Chapter 6

Conclusions and Future Directions

This thesis has contributed to the domains of sparse directory organization and private cache hierarchy design for server processors. This chapter summarizes the three contributions of this thesis and draws conclusions (Section 6.1). Next, it presents a few extensions and applications of the contributions (Section 6.2).

6.1 Summary and Conclusions

We present Pool Directory, a two-level sparse directory organization that optimizes the average number of bits devoted to track a block. The central microarchitecture innovation involves dynamic allocation from a pool of short vectors to build a tracking infrastructure for a block. The tracking infrastructure for a particular block can vary over time as the number of sharers of the block grows or shrinks. The tracking infrastructure may include just a first-level directory entry that includes one pointer, or a first-level directory entry and a second-level pool entry, or a first-level directory entry and a contiguous block of second-level pool entries. The fine-grain separation of the tracking needs based on sharer count allows the Pool Directory design to optimize the overall space allocated for tracking a block. For an equal space investment, the Pool Directory design performs better than the state-of-the-art designs while saving a significant volume of interconnection traffic.

Our Tiny Directory proposal is orthogonal to the Pool Directory design and addresses a much bigger concern related to the sparse directory organization. The Tiny Directory design shows a practically implementable way of operating with sparse directories that have a very small number of entries while offering performance that is within a percentage of a 2× directory. Three key innovations form the crux of the Tiny Directory design. First, a private block is tracked efficiently by borrowing a small part of the L3 cache block. Second, the critical subset of frequently shared blocks is tracked in a small sparse directory. Third, the critical shared blocks that cannot be tracked in the sparse directory due to space constraint are tracked in the L3 cache space, thereby offering a fall-back cushion to the Tiny Directory design. We demonstrate that a 128-core design can gracefully operate with Tiny Directories of size as small as $\frac{1}{256}\times$ while performing within a percentage of a 2× sparse directory. This is a significant leap in sparse directory height optimization compared to the state-of-the-art designs.

The efficiency of the private cache hierarchy is of great importance to the overall performance of a many-core processor. We show that a small group of read-shared data and code blocks generates a big portion of the interconnect traffic indicating that these blocks should be protected more carefully in the private cache hierarchy. Motivated by this observation, we replace the traditional mid-level (L2) cache of the private cache hierarchy by an exclusive L2 cache that serves as a victim cache of the L1 cache. The primary innovation in this design is the management of the L2 cache contents. We classify the L1 cache victims based on certain reuse distance and sharing degree features and on-the-fly estimate the collective reuse probability of each class. The L2 cache accommodates blocks from only those L1 cache victim classes that have reuse probability higher than a threshold. This technique automatically captures the important subset of the read-shared blocks that contribute to a significant portion of the interconnect traffic. Our proposal with a 64 KB L2 cache outperforms the baseline design having a 128 KB traditional non-inclusive/non-exclusive L2 cache and generates a lower volume of interconnect traffic, thereby saving a significant portion of the chip estate without losing any performance.

The three optimizations studied in this thesis are orthogonal to each other. The Tiny Directory proposal can use the Pool Directory design to further bring down the directory footprint while maintaining the same level of performance. The private cache optimization can be applied to any many-core server architecture irrespective of the directory organization. Specifically, both Tiny Directory and Pool Directory designs can be seamlessly combined with the private cache optimizations. Therefore, the three proposals are indeed orthogonal to each other and can seamlessly work together to realize the combined benefit of all three. The proposals together achieve one or more of the following important goals in a chip-multiprocessor with 100+ cores: saving in on-chip space investment, saving in interconnection traffic, improvement in performance.

6.2 Future Directions

We discuss four possible directions to extend the contributions of this thesis. First, the Tiny Directory proposal relies on a portion of the L3 cache block for tracking the privately cached blocks. This is easy to satisfy in the inclusive and non-inclusive/non-exclusive L3 cache designs. However, this becomes difficult in processors where the private blocks follow exclusion principle between the L3 cache and the private cache hierarchy e.g., in the AMD Magny Cours processor [22]. In this processor, blocks that do not have any past sharing history and are held privately inside the cores do not have any space allocated in the L3 cache. One way to overcome this difficulty is to track the private blocks that do not have a copy in the L3 cache using a multi-grain protocol i.e., each coarse-grain contiguous private region is tracked using a sparse directory entry. The rest of the blocks are tracked following the Tiny Directory proposal. It would be interesting to explore such a hybrid design for Magny Cours-like processors and evaluate how much the Tiny Directory footprint must increase to accommodate the coarse-grain region tracking entries without sacrificing any additional performance.

Second, the Tiny Directory proposal spills a carefully selected subset of the tracking entries into the L3 cache space. Extending this Tiny Directory + selective spill design to one extreme, one can imagine a design that does not have any sparse directory and the L3 cache space is dynamically shared between the blocks and their tracking entries, while a privately held block continues to get tracked in a portion of its copy in the L3 cache. The algorithms to manage the shared L3 cache space between program code/data and the tracking entries can offer interesting research directions. Such designs can be particularly attractive because no additional effort is needed to manage a separate sparse directory structure.

Third, we have designed the Tiny Directory for maintaining coherence within a single chip. It is not difficult to envision a similar proposal for optimizing the inter-socket coherence tracking overhead in a multi-chip or multi-socket setting. When Tiny Directory is applied to inter-socket coherence tracking, the logically shared main memory or shared L4 DRAM cache takes the place of the L3 cache. A block that is held privately within just one socket can be tracked by using a portion of the main memory or DRAM cache block. The tracking entries that cannot be accommodated in the Tiny Directory can be spilled into main memory or DRAM cache. If the L4 DRAM cache is non-existent or private to each socket, the Tiny Directory operations (e.g., spilling of tracking entries and using portion of data block for coherence tracking) will be done on the main memory, which is logically shared among the sockets. The interesting design challenge in such a proposal would be to make sure that the high latency of DRAM access does not lengthen the critical paths of coherence actions.

Fourth, we have shown that our private cache hierarchy proposal is only about 8% away from an optimal L2 cache design in terms of average L2 cache miss volume. This is still a reasonable gap and in some applications, the gap is larger. It would be interesting to further explore even more efficient L2 cache designs that aim at closing this gap.

Appendix: List of Publications

- Sudhanshu Shukla and Mainak Chaudhuri. Pool Directory: Efficient Coherence Tracking with Dynamic Directory Allocation in Many-core Systems. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD)*, pages 557–564, October 2015.
- Sudhanshu Shukla and Mainak Chaudhuri. Tiny Directory: Efficient Shared Memory in Many-core Systems with Ultra-low-overhead Coherence Tracking. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 205–216, February 2017.
- Sudhanshu Shukla and Mainak Chaudhuri. Sharing-aware Efficient Private Caching in Many-core Server Processors. In *Proceedings of the 35th IEEE International Conference on Computer Design (ICCD)*, pages 485–492, November 2017.

Bibliography

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A New Scalable Directory Architecture for Large-scale Multiprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 97–106, January 2001.

[2] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, **16**(1): 67–79, January 2005.

[3] S. V. Adve and K. Gharachorloo. Memory Consistency Models for Shared Memory Multiprocessors. *WRL Research Report 95/9*, December 1995.

[4] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280–289, May/June 1988.

[5] M. Alisafaee. Spatiotemporal Coherence Tracking. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture*, pages 341–350, December 2012.

[6] I. Atta, P. Tozun, X. Tong, A. Ailamaki, and A. Moshovos. STREX: Boosting Instruction Cache Reuse in OLTP Workloads through Stratified Transaction Execution. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 273–284, June 2013.

[7] I. Atta, P. Tozun, A. Ailamaki, and A. Moshovos. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 188–198, December 2012.

[8] I. Atta, P. Tozun, A. Ailamaki, and A. Moshovos. Reducing OLTP Instruction Misses with Thread Migration. In *Proceedings of the 8th International Workshop on Data Management on New Hardware*, pages 9–15, May 2012.

[9] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 421–432, December 2007.

[10] B. M. Beckmann, A. Basu, and S. K. Reinhardt. Dual-granularity State Tracking for Directory-based Cache Coherence. *United States Patent US8812786B2*, August 2014.

[11] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. In *IBM Systems Journal*, **5**(2): 78–101, 1966.

[12] 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*, pages 72–81, September 2008.

[13] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-grain Coherence Tracking. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 246–257, June 2005.

[14] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. In *IEEE Transactions on Computers*, **C-27**(12):1112–1118, December 1978.

[15] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[16] Y. Chang and L. Bhuyan. An Efficient Hybrid Cache Coherence Protocol for Shared Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 172–179, August 1996.

[17] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 293–304, September 2012.

[18] M. Chaudhuri and M. Heinrich. Exploring Virtual Network Selection Algorithms in DSM Cache Coherence Protocols. In *IEEE Transactions on Parallel and Distributed Systems*, **15**(8):699–712, August 2004.

[19] G. Chen. SLiD – A Cost-effective and Scalable Limited-directory Scheme for Cache Coherence. In *Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe*, pages 341–352, June 1993.

[20] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166, October 2011.

[21] J. H. Choi and K. H. Park. Segment Directory Enhancing the Limited Directory Cache Coherence Schemes. In *Proceedings of the 13th International Parallel and Distributed Processing Symposium*, pages 258–267, April 1999.

[22] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. In *IEEE Micro*, **30**(2):16–29, March/April 2010.

[23] J. D. Collins and D. M. Tullsen. Hardware Identification of Cache Conflict Misses. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 126–135, November 1999.

[24] B. A. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 93–104, June 2011.

[25] D. E. Culler, J. P. Singh with A. Gupta. "Parallel Computer Architecture: A Hardware/Software Approach".*Morgan Kaufmann Publishers Inc.*, 1998.

[26] S. Demetriades and S. Cho. Stash Directory: A Scalable Directory for Many-core Coherence. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*, pages 177–188, February 2014.

[27] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang. Building Expressive, Area-efficient Coherence Directories. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 299–308, September 2013.

[28] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 152–162, December 2011.

[29] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-core Systems. In *Proceedings of the 17th International Conference on High-Performance Computer Architecture*, pages 169–180, February 2011.

[30] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal Instruction Fetch Streaming. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 1–10, November 2008.

[31] A. Garcia-Guirado, R. Fernandez-Pascual, and J. M. Garcia. ICCI: In-cache Coherence Information. In *IEEE Transactions on Computers*, **64**(4): 995–1014, April 2015.

[32] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 81–92, June 2011.

[33] K. Gharachorloo, M. Sharma, S. Steely, and S. vanDoren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 13–24, November 2000.

[34] S. Guo, H. Wang, Y. Xue, D. Li, and D. Wang. Hierarchical Cache Directory for CMP. In *Journal of Computer Science and Technology*, **25**(2): 246–256, March 2010.

[35] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-based Cache Coherence Schemes. In *Proceedings of the International Conference on Parallel Processing*, pages 312–321, August 1990.

[36] J. L. Hennessy and D. A. Patterson. "Computer Architecture: A Quantitative Approach".*Morgan Kaufmann Publishers Inc.*, 2017.

[37] HP Labs. CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model. Available at http://www.hpl.hp.com/research/cacti/.

[38] HP Labs. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. Available at http://www.hpl.hp.com/research/mcpat/.

[39] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 209–220, May 2002.

[40] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely Jr., and J. S. Emer. High Performing Cache Hierarchies for Server Workloads: Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, pages 343–353, February 2015.

[41] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 60–71, June 2010.

[42] D. James, A. Laundrie, S. Gjessing, and G. Sohi. Distributed Directory Scheme: Scalable Coherent Interface. In *IEEE Computer*, **23**(6): 74–77, June 1990.

[43] N. E. Jerger, T. Krishna, and L-S. Peh. "On-Chip Networks".*Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers*, June 2017.

[44] N. P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, June 1990.

[45] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 34–45, April 1994.

[46] P. Kallurkar and S. R. Sarangi. pTask: A Smart Prefetching Scheme for OS Intensive Applications. In *Proceedings of the 49th International Symposium on Microarchitecture*, October 2016.

[47] C. Kaynak, B. Grot, and B. Falsafi. Confluence: Unified Instruction Supply for Scale-out Servers. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 166–177, December 2015.

[48] C. Kaynak, B. Grot, and B. Falsafi. SHIFT: Shared History Instruction Fetch for Lean-core Server Processors. In *Proceedings 46th International Symposium on Microarchitecture*, pages 272–283, December 2013.

[49] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel. WAYPOINT: Scaling Coherence to Thousand-core Architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 99–110, September 2010.

[50] S. M. Khan, D. A. Jimenez, D. Burger, and B. Falsafi. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 489–500, September 2010.

[51] H. Kim, B. Grot, P. V. Gratz, and D. A. Jimenez. Spatial Locality Speculation to Reduce Energy in Chip-Multiprocessor Networks-on-Chip. In *IEEE Transactions on Computers*, **63**(3): 543–556, March 2014.

[52] A. Kolli, A. G. Saidi, and T. F. Wenisch. RDIP: Return-address-stack Directed Instruction Prefetching. In *Proceedings of the 46th International Symposium on Microarchitecture*, pages 260–271, December 2013.

[53] R. Kumar, C-C. Huang, B. Grot, and V. Nagarajan. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture*, pages 493–504, February 2017.

[54] G. Kurian, O. Khan, and S. Devadas. The Locality-aware Adaptive Cache Coherence Protocol. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 523–534, June 2013.

[55] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. ATAC: A 1000-core Cache-coherent Processor with On-chip Optical Network. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 477–488, September 2010.

[56] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[57] Y. Li, R. G. Melhem, and A. K. Jones. Practically Private: Enabling High Performance CMPs through Compiler-assisted Data Classification. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 231–240, September 2012.

[58] P. Lotfi-Kamran, B. Grot, and B. Falsafi. NOC-Out: Microarchitecting a Scale-Out Processor. In *Proceedings of the 45th International Symposium on Microarchitecture*, pages 177–187, December 2012.

[59] P. Lotfi-Kamran, M. Modarressi, and H. Sarbazi-Azad. An Efficient Hybrid-Switched Network-on-Chip for Chip Multiprocessors. In *IEEE Transactions on Computers*, **65**(5): 1656–1662, May 2016.

[60] P. Lotfi-Kamran, M. Modarressi, and H. Sarbazi-Azad. Near-Ideal Networks-on-Chip for Servers. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture*, pages 277–288, February 2017.

[61] Y. Maa, D. Pradhan, and D. Thiebaut. Two Economical Directory Schemes for Large-scale Cache Coherent Multiprocessors. In *ACM SIGARCH Computer Architecture News*, **19**(5): 10–18, September 1991.

[62] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why On-chip Cache Coherence is Here to Stay. In *Communications of the ACM*, **55**(7):78–89, July 2012.

[63] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. In *IBM Systems Journal*, **9**(2): 78–117, 1970.

[64] S. S. Mukherjee and M. D. Hill. An Evaluation of Directory Protocols for Medium-scale Shared Memory Multiprocessors. In *Proceedings of the International Conference on Supercomputing*, pages 64–74, July 1994.

[65] H. Nilsson and P. Stenstrom. The Scalable Tree Protocol – A Cache Coherence Approach for Large-scale Multiprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 498–506, April 1992.

[66] B. O'Krafka and A. Newton. An Empirical Evaluation of Two Memory-efficient Directory Methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 138–147, May 1990.

[67] A. Ros and S. Kaxiras. Complexity-effective Multicore Coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 241–252, September 2012.

[68] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE Computer Architecture Letters*, **10**(1): 16–19, January-June 2011.

[69] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2012.

[70] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 187–198, December 2010.

[71] S. Shukla and M. Chaudhuri. Pool Directory: Efficient Coherence Tracking with Dynamic Directory Allocation in Many-core Systems. In *Proceedings of the 33rd IEEE International Conference on Computer Design*, pages 557–564, October 2015.

[72] S. Shukla and M. Chaudhuri. Tiny Directory: Efficient Shared Memory in Many-core Systems with Ultra-low-overhead Coherence Tracking. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture*, pages 205–216, February 2017.

[73] R. Simoni and M. Horowitz. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. In *Proc. of the International Symposium on Shared Memory Multiprocessing*, pages 72–81, April 1991.

[74] R. T. Simoni, Jr. Cache Coherence Directories for Scalable Multiprocessors. *PhD dissertation*, Stanford University, 1992.

[75] D. J. Sorin, M. D. Hill, and D. A. Wood. "A Primer on Memory Consistency and Cache Coherence". *Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers*, 2011.

[76] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–26, March 2013.

[77] P. Tozun, I. Atta, A. Ailamaki, and A. Moshovos. ADDICT: Advanced Instruction Chasing for Transactions. In *Proceedings of the VLDB Endowment*, **7**(14): 1893–1904, October 2014.

[78] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 335–344, September 2012.

[79] S. Volos, C. Seiculescu, B. Grot, N. K. Pour, B. Falsafi, and G. De Micheli. CCNoC: Specializing On-Chip Interconnects for Energy Efficiency in Cache-Coherent Servers. In *Proceedings of the Sixth International Symposium on Networks-on-Chip*, pages 67–74, May 2012.

[80] D. Wallach. PHD: A Hierarchical Cache Coherent Protocol. *Ph.D. dissertation*, MIT, September 1992.

[81] W.-D. Weber. Scalable Directories for Cache-coherent Shared-memory Multiprocessors. *Ph.D. dissertations*, Stanford University, January 1993.

[82] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[83] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[84] C-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr., and J. Emer. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 430–441, December 2011.

[85] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. SelectDirectory: A Selective Directory for Cache Coherence in Many-core Architectures. In *Design, Automation, and Test in Europe*, March 2015.

[86] J. Zebchuk, B. Falsafi, and A. Moshovos. Multi-grain Coherence Directories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, December 2013.

[87] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *Proceedings of the 42nd International Symposium on Microarchitecture*, pages 423–434, December 2009.

[88] L. Zhang, D. Strukov, H. Saadeldeen, D. Fan, M. Zhang, and D. Franklin. SpongeDirectory: Flexible Sparse Directories Utilizing Multi-Level Memristors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, August 2014.

[89] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan. SPATL: Honey, I Shrunk the Coherence Directory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 33–44, October 2011.

[90] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 135–146, September 2010.

[91] Y. Zheng, B. T. Davis, and M. Jordan. Performance Evaluation of Exclusive Cache Hierarchies. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 89–96, March 2004.