ABSTRACT

Title of dissertation: Studying the Impact of Multicore Processor Scaling on Cache Coherence Directories via Reuse Distance Analysis

Minshu Zhao, Doctor of Philosophy, 2015

Dissertation directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

Directories are one key part of a processor's cache coherence hardware, and constitute one of the main bottlenecks in multicore processor scaling, *e.g.* core count and cache size scaling. Many research effects have tried to improve the scalability of the directory, but most of them only simulate a few architecture configurations. It is important to study the directory's architecture dependency, as the CPUs continue to scale. This is because besides applications, directory behaviors are also highly sensitive to architecture. Varying *core count* directly affect s the amount of sharing in the directory, and varying the *data cache hierarchy* affects the directory access stream. But unfortunately, exploring the huge design space of multiple core counts and cache configurations is challenging using traditional architectural simulation due to the slow speed of simulations.

This thesis studies the directory using multicore reuse distance analysis. It extends existing multicore reuse distance techniques, developing a method to extract directory access information from the parallel LRU stacks used to acquire private-

stack reuse distance profiles. This thesis implements this method in a PIN-based profiler to study the directory behavior, including the directory access pattern and directory content, and to analyze current directory techniques.

The profile results show that the directory accesses are highly dependent on cache size, exhibiting a 3.5x drop when scaling the data cache size from 16KB to 1MB; the sharing causes the ratio of directory entry to cache blocks to drop below 50%; and the majority of the accesses are to a small percentage of the directory entries. Cache simulations are performed to validate the profiling results, showing the profiled results are within 14.5% of simulation on average. This thesis also analyzes different directory techniques using the insights from the profiler. The case studies on the Cuckoo, DGD, SCD techniques and multi-level directories show that required directory size varies significantly with CPU scaling, the opportunity of compressing private data decreases with cache scaling, reducing the sharer list size is an effective technique and a small L1 directory is sufficient to capture most of the latency critical accesses respectively.

Studying the Impact of Multicore Processor Scaling on Cache Coherence Directories via Reuse Distance Analysis

by

Minshu Zhao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor Donald Yeung, Chair/Advisor
Professor Manoj Franklin
Professor Bruce Jacob
Professor Ankur Srivastava
Professor Alan Sussman

Dedication

To my parents, who have been always there for me.

Acknowledgments

First, I would like to thank my advisor, Dr. Donald Yeung, for his patience, encouragement and immense knowledge. His advice and guidance helped me all the way from taking courses through writing this thesis. He made the last six years a rewarding journey.

My thanks also goes to the members of my committee, Dr. Franklin, Dr. Jacob, Dr. Srivastava and Dr. Sussman, for their insightful comments, hard questions and extreme patience.

I also would like to thank my fellow students in the lab, Meng-Ju Wu, Inseok Choi, Michael Badamo and Jeff Casarona for their support, feedback, suggestions and friendship. In particular, I am grateful to Dr. Meng-Ju Wu, who laid the foundation for this work, enlightened me the direction of research and gave me many insightful discussions.

Finally, I would like to thank my parents, Yunyi Zhao and Chao Zhu, who encouraged me to pursue Ph.D. study in the first place and supported me all the way through my study emotionally.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

High-performance microprocessors are integrating an increasing number of cores on chip. Today, CPUs with 8–10 state-of-the-art cores or 10s of smaller cores [1,2] are in the market. For example, Knights Landing announced by Intel will be built using up to 72 Airmont (Atom) cores with four threads per core [3]. CPUs with 100s of cores–*i.e.*, large-scale chip multiprocessors (LCMPs) [4,5]–will be possible in the near future. This offers enormous potential performance improvement for programs with thread-level parallelism.

At the same time, new memory technologies have been adopted to provide more storage close to the cores, thus keeping up with the compute scaling. For example, Knights Landing employs on-package eDRAM [3]. There are also studies on Phase-Change RAM [6], Spin Transfer Torque RAM [7] and Resistive RAM [8]. These new technologies will provide the potential for much larger amounts of cache on chip.

While LCMPs are growing with more cores and larger on-chip caches, to fully utilize them, computer architects face huge challenges in scalability. Among these challenges, one of the most critical is cache coherence. Snooping protocols, which send cache coherence operations through a bus, only work for small-scale sys-

tems. When scaling beyond 16 or 32 cores, a directory-based protocol is required. Directory-based protocols keep a separate directory that stores the state and sharers of the cache blocks it tracks for coherence, enabling point-to point protocols that are more scalable than snooping protocols. But scaling this directory beyond hundreds of cores and hundreds of megabytes of cache is a key problem.

Ideally, a directory should consume small area and power, and incur low latency overhead when scaling to large core count and cache size. A directory's area, latency overhead and power can be tuned via a few design parameters. First, the area of a directory is determined by its capacity. Second, the latency overhead of a directory can be calculated by latency per access times its access frequency. Lastly, power dissipation of a directory is composed of static power, which is determined by its capacity, and dynamic power, which can be calculated by the energy per access times access frequency. In addition, both latency per access and energy per access of the directory are affected by the directory's capacity and architecture, while the access frequency of the directory is mostly decided by the data cache capacity. Therefore tuning *directory capacity* and *directory architecture* can change the balance between area, latency and power in the directory.

Many directory architectures have been proposed to make different trade-offs in balancing these three design parameters. Duplicate tag directories [9] are area-efficient, but require high associativity as CPUs scale. This architecture incurs large overhead in access latency and energy per access, thus becoming not scalable with a large number of cores. Sparse directories [10], on the other hand, are more scalable, because they store the sharer list per cache tag in a cache with low associativity.

Unfortunately, the sparse directory architecture can become very big, as its size scales with both core count and cache size. When the cache size increases, there are more tags that need to be stored in the directory, whereas when the core count increases, there are more sharers that need to be tracked by each directory entry. For example, a full-map sparse directory for a 256-core CPU (assuming 64-byte cache blocks) will be half the size of its associated cache. Moreover, sparse directories require over-provisioning the number of tags to avoid conflicts due to their low associativity, further worsening the area and power requirement of a sparse directory as cache capacity scales.

Researchers have developed many approaches to improve the capacity scaling of directories. One approach is to reduce the sharer lists, such as using alternative sharing representations [10–15], or hierarchical directories [16–18]. Another approach is to reduce conflicts [18–20]. There are also approaches exploiting private data [20–23]. However, these techniques do not come without a price; many of them increase the complexity of the directory and/or access to the directory, complicating the directory design.

The balance among area, power and performance is not only affected by the directory techniques, but it is also affected by directory behavior, such as the access frequency to the directory and the content of the directory. Therefore, to complete the picture of the effectiveness and trade-off of different directory techniques, directory behavior needs to be studied–specifically, the *directory access patterns* and *directory content characteristics*. *Directory access patterns* include the read/write access frequency, as well as access distribution over different directory entries. This

information helps designers to understand the cost of each technique. *Directory content characteristics* include the total number of entries in the directory, as well as the degree of sharing for each entry. They also include the type of sharing–read vs. write. This information helps designers to determine how much the directory can be compressed.

These two important factors are impacted by how applications exercise the directory. For example, sharing is inherently an application behavior, with any sharing that occurs in the directory traceable to the interaction between application threads. Also, the directory access streams are dependent upon the memory access streams, which are decided by applications too.

But in addition to applications, directory content and access patterns are also highly sensitive to architecture. For example, varying *core count* usually changes the number of application threads, directly affecting the amount and frequency of sharing, thus changing the content of the directory. Also, varying the *data cache hierarchy* affects directory access streams because the directory access streams are cache-filtered versions of the memory access streams. Therefore, the cache size will affect the directory behavior by changing directory access streams.

On the other hand, the data cache hierarchy can also alter the sharing captured by the directory. This is because only the sharing that occurs in the private caches is visible to the directory, while the sharing that occurs in the shared cache or main memory is not visible. And the data cache size affects where the sharing happens. For example, the sharing that happens between two application threads may occur far apart in time. When the data cache is small, it is possible that the data block is

evicted from the private cache before the second access happens. In this case, the sharing of the data block does not go through the private cache, thus it is invisible to the directory. But if the data cache is large enough to retain the data block, then the sharing of this data block happens in the private cache, and the directory is needed to provide coherence information. Therefore, the data cache size affects the directory behavior by changing the sharing pattern as well.

Given the importance of memory coherence to multicore scalability, it is crucial to study different directory techniques and their application and architecture dependencies. Traditionally, computer architects have used architectural simulation alone to study directory effects. Simulators can model memory behavior accurately, but simulating CPUs with 100s of cores is extremely slow. Moreover, one simulation only represents one individual architecture configuration and input problem. Simulation sweeps are usually required to gain deep insights. But with increasing number of cores and more complex cache hierarchies, the design space for the directory is growing exponentially large. Therefore, many researchers only vary the application, *i.e.*, by running entire benchmark suites, but they do not vary the cache configuration when studying directory techniques. There are only a few studies that have simulated different core counts or cache sizes [19, 22, 24]. And even in their cases, they only look at a small number of configurations.

One of the tools that can help architects evaluate multicore caches is *reuse distance (RD) analysis* [25–30]. RD analysis evaluates cache hierarchies using *RD profiles*, which capture program-level locality. Recently, private-stack reuse distance (PRD) profiling [27–30] has been proposed to model the interaction in private data

caches using per-thread coherent LRU stacks.

RD profiles are *architecture independent* across cache scaling, *i.e.*, a few profiles can reveal cache behavior across a large number of CPU configurations. Studies also show that PRD profiles for programs with symmetric threads are essentially architecture independent across cache size scaling as well [26, 27, 30, 31].

In light of this, this thesis applies multicore RD analysis to study the directory behavior. As explained above, the directory behaviors as well as the trade-off of different directory techniques are architecture dependent. This thesis extended RD analysis to provide a fast way to study the directory behavior across different CPU configurations, giving insights into the directory scalability problem, similar to what RD analysis has provided for the data cache. A framework is proposed based on PRD stacks that can extract the directory access and sharing information. In particular, *relative reuse distance between sharers* is proposed in this thesis. Relative reuse distance quantifies the sharing distance between accesses, identifies the sharing that occurs in the private cache, and thus directory, and enables the capacity-sensitive directory behavior analysis. Then, insights of directory access and sharing patterns are used to study the effectiveness and trade-off of directory techniques when core count and/or cache size scale.

The analysis is implemented in a PIN-based profiler [32] to study directory behavior when scaling cache size and core count. Three aspects of the directory behavior are studied in this thesis: directory access frequency, directory contents and directory access distribution. For directory access frequency, the profiling results show a 3.5x drop in total accesses when increasing cache sizes from 16KB to 1MB,

despite an increase in sharing-based directory accesses. For directory contents, the results show an increase in number of shared entries and a reduction in private entries when scaling cache sizes, enabling an reduction in total number of directory entries. The results show that directory size can be reduced by 53.3% in terms of coverage. For directory access distribution, the results show that at 1MB, 23.0% of the entries receive 37.8% of the total directory accesses and 82.7% of the sharing-based directory accesses. The profiling results also show that core count scaling has a much smaller effect on directory than cache size scaling. With 64MB total cache, the directory accesses increases by 38% and the directory size increase by 2.3% despite a 16x increase in core count.

To validate the profiling results, cache simulations are performed in this thesis to compare against profiling results. The validation results show the profiled directory accesses are within 8.6% of simulation on average across cache size scaling, and within 12.2% of simulation on average across core count scaling. Moreover, the profiled directory coverage results are within 11.2% of simulation on average across cache size scaling, and within 14.5% of simulation on average across core count scaling. In addition, the error of profiled directory access distribution results is 8.7% on average across cache size scaling, and 8.1% on average across core count scaling.

This thesis also discusses the implications of the profiling results for current directory techniques. First, the fraction of on-chip memory for directory varies with cache size scaling. Experiments show that for most benchmarks, a Cuckoo directory only needs to provide entries for 37.5–87.5% of the cache blocks in the private caches. Second, the fraction of shared entries increases with cache size scaling. Experiments

show that the average reduction for DGD technique compared to a regular directory decreases from 49.6% to 26.7% with cache size scaling. Third, most entries exhibit a limited sharing degree even with cache size scaling. Experiments show that on average, the increase in directory entries for the SCD technique is within 7.2%. Fourth, a small fraction of the directory receives a large portion of the directory accesses. Experiments show that in a multi-level directory, the first level that only covers 18.75% of the cache blocks in the private caches receives 83–91% of the latency critical directory accesses. Lastly, there are more opportunities to trade off directory access latency for directory size with cache size scaling, indicating the overheads for the above techniques reduce with cache size scaling.

The rest of this thesis is organized as follows. Chapter 2 provides the background on cache coherence structures, directory and the current directory techniques. Chapter 3 provides the background on sequential and multicore reuse distance analysis, and previous work studying cache sharing behavior using RD analysis. Chapter 4 discusses how reuse distance is used in analyzing directory behavior. Chapter 5 discusses the detailed implementation of the analysis framework. Chapter 6 reports profiling results while chapter 7 validates the profile results with cache simulations. Chapter 8 discusses implications and the case studies. Finally, chapter 9 concludes the thesis.

Chapter 2: Background on Directory

2.1 Cache Coherence and Directory

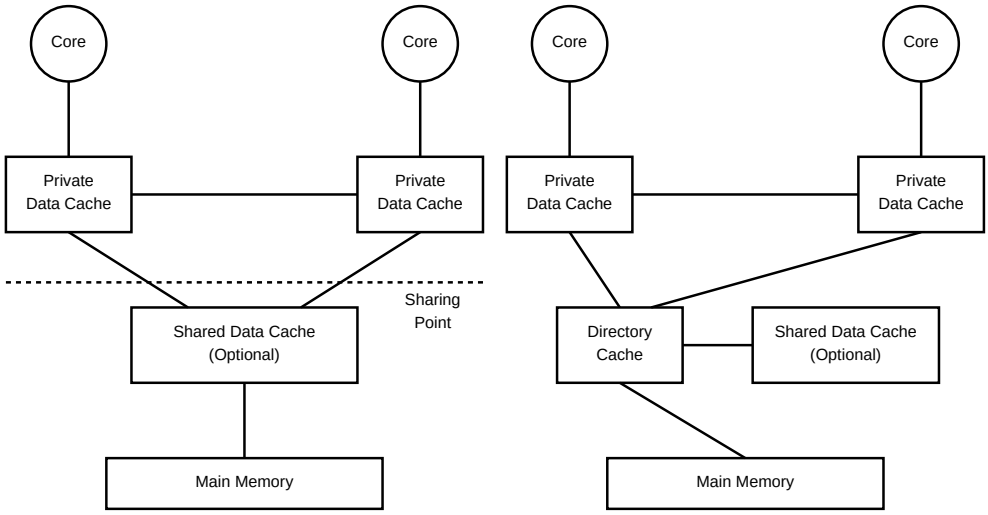
The memory hierarchy of a typical multicore usually consists of multiple levels of private caches, an optional shared cache, and a shared main memory, as illustrated in Figure 2.1(1). Cores and their private data caches sit at the top of the hierarchy, with multiple levels of private cache per core. Off-chip memory sits below the cache hierarchy.

Caching shared data in the private caches introduces a cache coherence problem. This is because the cores see memory through their local private caches, and without precautions, they can see different value of same memory location in their respective caches [33].

Hennessy and Patterson discussed the three aspects of cache coherence in the book *Computer Architecture, Fourth Edition: A Quantitative Approach* [33].

1. Preserving program order. After a write by core C to memory location X, a read by C always returns the value written by C, if there is no writes to X by another core between the write and the read by C.

2. Cores should not continuously read an old data value. After a write by one



(1) Multicore Cache Hierarchy (2) Directory Accesses

Figure 2.1: Directory Accesses in a Multicore Cache Hierarchy

core to memory location X, a read by another core to location X should return the written value, if the read and write are sufficiently separated in time and no other writes to X occurs between the two accesses.

3. Writes to the same memory location are serialized. That is, if value 1 and then 2 are written to one location, cores can never read the value of location as 2 and then later read it as 1.

Also as explained above, cache coherence is only relevant to the private caches, not the shared structure. Therefore this thesis defines the line between last level private caches and the first shared structure (either shared cache or main memory) as the CPU's *sharing point*. Cache coherence has to be maintained above the *sharing point* while there is no coherence issue below the *sharing point*.

2.1.1 Directory

The protocols to maintain coherence for multiple processors are called cache coherence protocols. The cache coherence protocol keeps track of the state of any sharing of a data block, *e.g.*, whether the block is shared or private, and if the former, the cores that are sharing the data block. Traditionally, there are two ways to maintain cache coherence between multiple cores.

One is the snooping protocol, which sends cache coherence operations through some broadcast medium, such as a bus. In snooping protocols, each cache that has a copy of the data also has the sharing status of the block, but there is no centralized structure for this information. The request of a cache block is sent to the broadcast medium and the broadcast medium is snooped by the cache controller of each core, to determine if the core has a copy of the data that is being requested.

Snooping protocols are simple and do not require significant hardware support, because they rely on a single point of serialization: the broadcast bus. But they are very message intensive. When the number of cores increases, snooping protocols will generate too many messages for the bus to handle. Therefore the snooping protocol has limited scalability.

The alternative to snooping protocols is to keep a directory. A directory-based protocol keeps a separate directory that stores the state of the block it tracks for coherence. Each entry contains sharing information of a memory block, such as whether the block is dirty or which caches have a copy of this block. The use of a directory structure can reduce the bandwidth demands, as memory requests only

need to go to the directory, not to every other core in the CPU.

One simple directory implementation is to attach each memory block with the directory entry that is in the shared structure, either shared data cache (assuming inclusive) or main memory. This is taking advantage of the shared structure as one single point for accesses to be serialized. It is also called an in-cache directory if the directory is implemented in the shared cache. In such an implementation, the amount of the information is proportional to the size of the shared structure.

An in-cache directory can be very costly in terms of area. Because the shared cache is usually much bigger than the last level private cache, not all the cache blocks in the shared cache are cached in the last level private cache. These blocks that are not cached by the private caches need neither coherence nor directory entries. Therefore, when the directory is integrated with shared cache, it contains many empty entries, wasting precious on-chip area. For example, modern day Intel microprocessors (like the Intel Core i7 processor) implement inclusive last level shared caches, and these are 8x the size of the cores' private caches [34]. If an in-cache directory is implemented, then at least 87.5% of the directory entries are not in use at any given time.

On the other hand, for a directory that is implemented in main memory, the overhead is not only the area, but also the long access latency. Because many directory accesses involve the data that are already on chip, accessing main memory before proceeding greatly increases the access latency of such accesses.

To reduce the area overhead, some approaches try to only track the active cache blocks that are resident in the private data caches and put this smaller directory

cache on chip. One of the approaches is duplicate-tag-based directory [9, 35]. A duplicate-tag-based directory uses the existing tags in the private cache and stores all of them in central place, usually sitting at the CPU's sharing point. Therefore, the state for a block can be determined by examining the directory to find out all the copies of the tag in every possible cache.

Duplicate-tag-based directory is simple to implement and requires low area cost. However, its associativity increases proportional to the core count, which makes it not scalable. To illustrate, with a 64-core CPU, the directory needs to contain the tags for all 64 possible cache locations. And if each cache is 16-way set associative, then the aggregate associativity of the directory is 1024 ways.

Another approach is using a sparse directory [10]. The sparse directory organizes the cache coherence information as a cache. Each entry of the sparse directory is indexed by the tag of the blocks that are active in the private cache, and the entry contains the sharing information. Similarly, a sparse directory also sits at the CPU's sharing point. Figure 2.1(2) illustrates a directory cache with the on-chip cache hierarchy.

Because the sparse directory is implemented in a similar way as a cache, the architect is free to decide the associativity of the directory, making it more scalable. However, there is area overhead associated with the sparse directory. One of them is over-prevision. Theoretically, the directory cache size can be the same as the total number of private cache blocks, but a sparse directory is usually over-provisioned by 2x to 4x to reduce the probability of conflicts in the directory, in case the memory access pattern is skewed to load one directory set more heavily than the others [10].

With increasing size of the private data cache, the over-prevision of the directory can cost a lot of on-chip area.

Moreover, a straightforward way to implement a directory entry is to use a bit vector with one bit per core, also known as a full map entry. Therefore, a system with N cores requires N bits per entry. The directory entry scales linearly with the core count. As the core count increases, full map can incur significant area overhead too. To illustrate, a 256-core CPU needs 256 bits (32 bytes) for every directory entry. And if the directory is over-provisioned by 4x, for each 64-byte cache entry, 128-byte directory capacity is needed to maintain the coherence, making the directory larger than the caches. The huge area requirement for high core count makes sparse directory unscalable too. Therefore, many researchers have proposed to reduce the directory size and thus improve the scalability. This thesis discusses various techniques in detail in Section 2.2.

2.1.2 Cache Coherence Protocol

To maintain the coherence requirements discussed above, a read should always see the value of the most recently write. Hennessy and Patterson discussed the two types of protocols to maintain the cache coherence in the book *Computer Architecture, Fourth Edition: A Quantitative Approach* [33]. One is write update protocol. The write update protocol updates all cached copies of a data block when the data is written to. However, because such protocols have to multicast the writes to all existing copies, the bandwidth requirement for this protocol is huge. Therefore,

this protocol is not as widely used as the alternative protocol, the write invalidate protocol.

The write invalidate protocol invalidates copies of a data block when a write happens, so that the core that issues the write has exclusive access to the data before it writes to the data. This protocol is the most common protocol for directory-based schemes, so this thesis will focus on write invalidate protocols.

In addition to invalidating all the other copies of a cache block that is being written to, a cache miss should always find the most recent value of the data. There are two types of cache regarding write policy, write through and write back. In a write-back cache, the most recent copy is in the cache, but not in the memory. Therefore, the directory has to provide the information of where the most recent copy is. Because write-back cache requires less memory bandwidth, it is widely used in multicore CPU, so this thesis will focus on write invalidate protocols with write-back caches.

2.1.2.1 MESI protocol

The MESI protocol is a widely used cache coherence protocol developed by the University of Illinois at Urbana-Champaign [36]. This section explains the protocol in detail because the analysis in this thesis focuses on MESI protocols.

MESI identifies the four states in which a cache block can be in:

- Modified state. This indicates that the block has been modified in the cache. Therefore the data in the cache is the most recent copy and is inconsistent

	Mode	Hit or Miss	State	Directory Access	Comment
1	R	miss	I -> E	Create Directory Entry	Data From Next Level
2	R	miss	M,E,S -> S	Directory Entry Access	Data Forwarding
3	R	hit	M,E,S	No Directory Access	Data Cache Hit
4	W	miss	I -> M	Create Directory Entry	Data From Next Level
5	W	miss	M,E,S -> M	Directory Entry Access	Data Forwarding, Invalidations
6	W	hit	S -> M	Directory Entry Access	Invalidations
7	W	hit	M,E -> M	No Directory Access	Data Cache Hit

Table 2.1: Access mode, cache hit or miss, cache block state, directory access and comments for MESI protocol.

to the data below the sharing point. Also the core which owns this data has exclusive access to it. The eviction of this block will cause a write back to the memory.

- Exclusive state. This indicates that the block is only in one is cache, but unmodified. This state can be changed into Shared state in response to a read request, or changed into Modified state when being written to. The Exclusive to Modified change can be done locally without notifying the directory.

- Shared state. This indicates that the block is unmodified and exists in more than one private cache. The block cannot be written to in this state. Also the eviction of this block does not need a write back to the memory.

- Invalid state. This indicates that the block is not in any of the private caches.

Table 2.1 shows cache coherence mechanism based on the access mode, hit or miss and the state of the cache block. For example, a cache block starts out in with invalid state, and a read miss or write miss will bring the cache block into the private cache, and create a new directory entry for it. A read miss puts the block

into exclusive state, while the write miss puts the block into modified state. While the block is valid in the private cache, a transition to shared state from modified or exclusive state is triggered by a read miss from another core. This read miss will update the directory and also write back the data to the next level. On the other hand, a transition to the modified state from shared state is triggered by a write. This write will access the directory to send out invalidation messages.

As Table 2.1 shows, different cache transactions can trigger different directory accesses. Section 4.2 will discuss this information can be used to study the behavior of the directory in detail.

2.1.2.2 MSI protocol

The MSI is a basic cache coherence protocol. Compared to the MESI protocol, the MSI protocol does not have an Exclusive state. In the MSI protocol, shared state indicates the block is unmodified and exists in *one or more* private caches. Therefore, the exclusive state in MESI protocol is a special case of shared state in MSI protocol.

Having exclusive state helps to reduce one type of directory access, the transition form exclusive state to modified state. In MSI protocol, such transition is from shared state to modified state, and incurs a directory access. Therefore the MSI protocol requires more memory bandwidth than the MESI protocol.

2.1.2.3 MOESI protocol

The MOESI protocol is another popular cache coherence protocol proposed by Sweazey and Smith [37]. In addition to the four states in the MESI protocol, MOESI adds an Owned state.

- Owned state. This indicates that the block is both modified and shared. This state is triggered when another core issues a data request to a block in modified state. Instead of changing to shared state and writing back the data as in the MESI protocol, the data is forwarded to the requesting core, changes into owned state without writing back to the next level.

The owned state avoids the need to write back a modified data to next level before sharing it. Therefore, the write-back is deferred and in some cases, multiple write-back can be combined into one write-back, thus saving memory bandwidth.

Therefore, depending on the cache coherence protocol, the access frequency to the directory and the behavior of the directory can be different. Section 4.4 discusses how different cache coherence protocols change the directory behavior.

2.1.2.4 Protocol Variation in Evictions

Apart from different cache coherence protocols, another important part of the directory protocol implementation is how the eviction of data is handled. In a write-back cache, when a modified block is evicted from the data cache, it will write back to the next level memory structure. And naturally, the directory needs to be notified

because the location of the most recent copy of this data is changed. However, for a evicted data block that is clean, it is unclear whether the directory needs to be notified. Notifing the directory will cause extra traffic on chip, but this also enables the directory to update the sharing information. For example, in a sparse directory, by enabling clean eviction notification, a directory entry knows the exact number of existing copies of the corresponding data in the private caches and can be evicted if the number goes down to zero.

The difference in handling the data eviction in different directory techniques affects the analysis in this thesis, and will be explained in Section 4.4.

2.2 Previous Directory Techniques

As explained in Section 2.1.1, the size of the sparse directory is the major problem in scalability. A number of prior works have explored many ways to reduce the size of directories, and this thesis studies the characterization of directory behavior and evaluates the effectiveness and the cost of prior works.

Several works have focused on *reducing sharer lists*. One way is to use a compressed but inexact encoding for each entry. Gupta et al. proposed the coarse vector technique, which uses N/K bits, where a bit is set if any of the sharers in a K-sharer group caches the block [10]. Acacio et al. proposed sharing code based on multilayer clustering approach [38]. It constructs the nodes logically in a tree structure and the sharing code is the level of the root of the minimal sub-tree that contains the home node and all the sharers.

Another approach is to maintain the exact sharer lists. IEEE Standard Scalable Coherent Interface (SCI) [39] proposed chained directory protocol. Each directory entry contains a chain of pointers to all the sharers. Nilsson and Stenstrom proposed a balanced tree structure instead of a link list for each entry [40] and Chang and Bhuyan improved it by maintaining multiple trees for each entry [41]. A hierarchical directory is also proposed to implement multiple levels of directory cache, in which each first-level directory encodes the sharers of a subset of caches, and the higher level tracks the directory of lower level caches [16–18, 42]. Recently, Zhao et al. observed that many memory locations have the same sharer list and thus proposed SPACE [15]. SPACE encodes sharing patterns in the directory entry and has a separate table to decode the pattern into bit-vectors.

Moreover, one observation is that in many benchmarks, only a few memory blocks are widely shared among all cores. This has enabled researchers to propose a directory entry format that only tracks a few sharers. Agarwal et al. evaluate the schemes of limited pointers, in which a smaller number of pointers is used to identify the sharers [11]. Choi and Park proposed the segment directory, a hybrid of limited pointers and bit-vector, to further reduce the entry size [12]. Each directory entry is of size $K+\log_2 N/K$ and can track at most K sharers. The problem of the limited pointer technique is sharer overflow, when the number of pointers is not enough to track all the sharers. Agarwal et al. evaluated the broadcasting policy as well as the no-broadcast policy in which extra sharers get invalidated [11]. Chaiken et al. proposed a software fallback, where the software emulates a full map directory when overflow happens [14]. Researchers have also combined limited pointers with

various techniques mentioned above. Gupta et al. proposed switching to coarse representation [10] and Chen proposed using chained pointers when directory entries overflow [13]. Sanchez et al. proposed SCD [18] to scale up to a thousand cores or more. For cache blocks with narrow sharing, SCD uses pointers in entries, while for blocks with wide sharing, it uses a hierarchy of multiple directory entries.

The capacity reduction of these techniques is dependent upon how the directory entry is encoded, but the overhead is related to directory behavior. When the techniques employ complex directory protocol, they usually require multiple lookups per directory access, increasing the directory access latency. This thesis analyzes the directory access frequency, thus it can estimate the cost of such techniques.

Apart from reducing the directory entry size, there are techniques that focus on reducing the number of entries. One is to reduce the over-provisioning by *reducing conflicts.* Ferdman et al. proposed the Cuckoo Directory [19] to reduce conflict misses using Cuckoo Hashing [43]. ZCache-style replacement [44] is also used to reduce the conflicts in SCD [18] and DGD [20]. The directory content studied in this thesis shows a minimum number of directory entries needed in these techniques assuming full associativity; therefore, it gives a lower bound on the directory capacity when designing a directory cache using these techniques. On the other hand, these techniques combine sophisticated hashing schemes and re-insertions, increasing the cost of directory accesses unevenly. This thesis studies the accesses frequency for different types of directory accesses, thus it can help determine the cost of these techniques.

There are also approaches focusing on *exploiting private data* to reduce the

number of entries. Cuesta et al. use the operating system to detect private pages and omit tracking cache blocks in those private pages [23, 45]. Valls et al. proposed PS-Dir [21], which stores shared cache entries in regular SRAM, while keeping private entries in slower eDRAM. SCT [22] and DGD [20] recognize private data accessed by each core tend to occur in large contiguous regions. Hence, they coalesce consecutive privately accessed cache blocks, and track them as a single coherence unit. The directory content analysis in this thesis can determine the number of shared entries in the directory, thus giving a lower bound of these techniques. The analysis can also be performed assuming different cache block sizes, giving an estimate of their capacity reduction. In addition, these techniques require complicated directory operation, therefore the directory access pattern analysis in this work can be helpful in analyzing the cost of such techniques.

The tagless coherence directory [46] proposed by Zebchuk et al uses bloom filters to track which blocks are in the private cache. This approach significantly reduces directory storage requirements because the tags are not stored, so the capacity of the directory purely depends on the number of bloom filters. However, the approach uses multiple hash functions, and access energy can be very high. The directory access frequency analysis in this thesis can help determine whether it is energy-scalable.

Finally, the last technique is to exploit the locality of coherence operation. The thesis shows that not all entries in the directory are equally important. Some directory entries are accessed much more frequently than other directory entries WayPoint [24] proposed a two-level directory. It evicts infrequently accessed entries

that do not fit in the on-chip directory, and stores them in off-chip DRAM. The access distribution analysis in this thesis can help analyze this kind of non-uniform access distribution across directory entries.

Chapter 3: Background on Reuse Distance Analysis

3.1 Reuse Distance Profile

Reuse distance (RD) analysis is a tool to characterize cache behavior by analyzing program locality. It is initially developed for uniprocessor to study the locality of the program. The analysis measures an *RD profile*, which records RD values for all memory references.

An RD value is defined as the number of unique data blocks referenced since the last reference to the same data block. To illustrate, Figure 3.1(a) shows a memory access stream for uniprocessor accessing seven memory blocks, $A - G$. At $t = 5$, block A is accessed, the RD value for this access is 3 because there are three unique references between this access and the previous access to block A at $t = 1$.

An LRU stack is used to obtain the RD values. The stack contains the program's memory blocks, and is maintained like a cache with LRU ordering among the blocks. When a memory access is performed, the LRU stack is searched to find

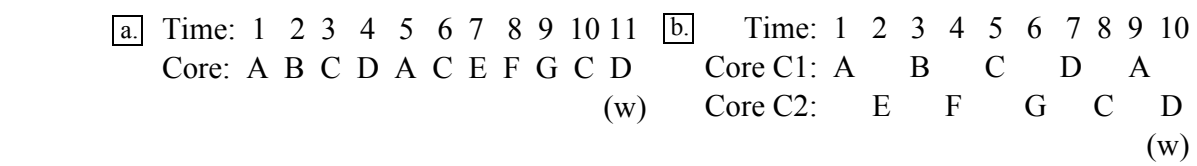


Figure 3.1: memory reference streams for uniprocessor and multicore

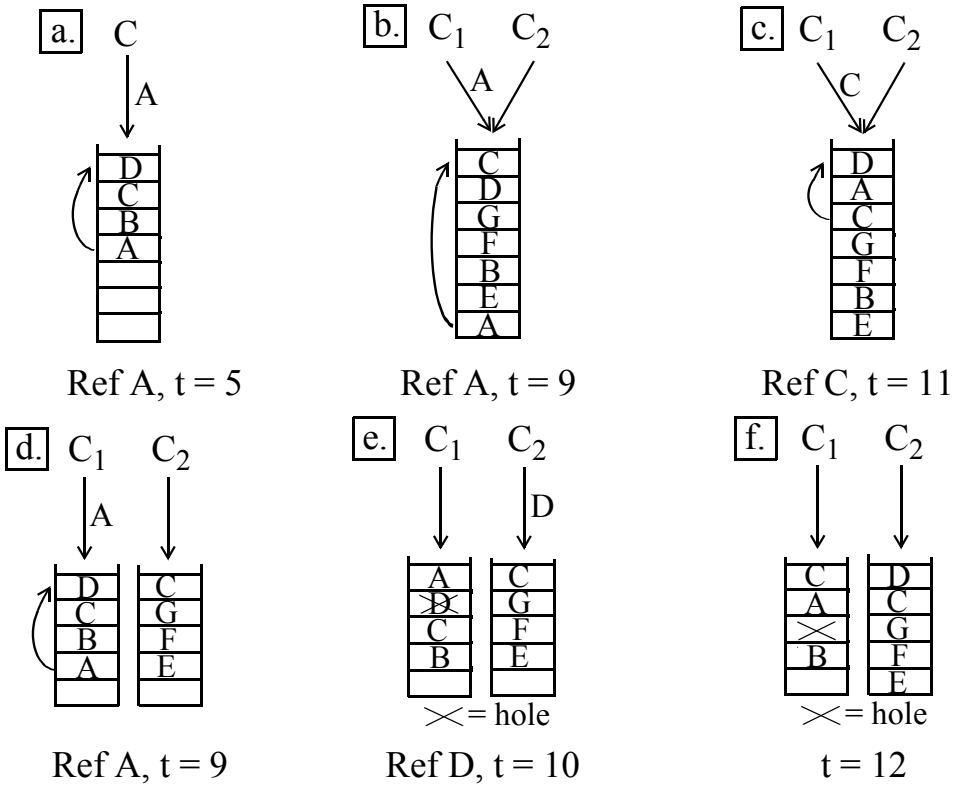


Figure 3.2: LRU stacks for RD, CRD, and PRD

the memory block and the stack depth of the block is its RD value. Then the entry for this memory block is moved to the top of the stack. To illustrate, Figure 3.2(a) shows the state of the LRU stack at $t = 5$, the stack depth of block A is 3, which is the RD value for this access.

Moreover, assuming LRU and full associativity, a cache of size CS can satisfy any reference with RD $< CS$; therefore the cache miss count for this cache is the sum of all references in the RD profile with RD $\geq CS$. To illustrate, Figure 3.2(a) shows the state of the LRU stack at $t = 5$, and shows the RD value for this access to block A. Therefore, this access will be a hit in a cache with 4 or more entries, but a miss in a cache with less than 4 entries.

In sequential programs, the RD profiles are architecture independent, because

the memory access stream is fixed once the binary of the program is produced. Each program produces one memory access stream results in one RD profile. With this profile, it is possible to predict the misses for any cache size CS, and evaluate cache performance at any cache size CS. Therefore RD profiles allow fast analysis along the cache capacity dimension.

In recent years, RD analysis is extended for multicore processors by modeling *inter-thread interactions*. The memory hierarchy in multicore processor usually contains two different cache architecture, shared cache and private cache. To evaluate shared cache, *Concurrent reuse distance (CRD) analysis* [25–31] is developed. It uses one global LRU stack to model the reuse across thread-interleaved memory references. To evaluate private cache, *private-stack reuse distance (PRD) analysis* [27–30] is developed. It models multiple LRU stacks, one stack per core, and the memory references are accessed on its local stack while the coherence is maintained among all stacks.

RD analysis for multicore CPUs is more complex compared to the sequential RD analysis because the locality in parallel programs involves both per-thread reuse and memory reference interaction between simultaneous threads. For example, in a shared cache, inter-thread memory reference interleaving leads to *interference*. While in private caches, *data sharing* leads to replications and communications.

Moreover, memory interleaving caused by inter-thread interactions depends on timings. Therefore RD analysis for multicores become sensitive to how the memory accesses are interleaved, *i.e.*, *architecture dependent*. CRD/PRD profile measured on one architecture might not be valid for another architecture if the per-thread

timing changes substantially, unable to reflect the locality for the cache accurately.

However, studies show that in programs whose threads execute similar codes, *e.g.*, programs that exploit *loop-based* parallelism, the memory interleaving is regular [26, 27, 30, 31]. Because these threads tend to speedup or slow down by similar amounts with different architectural configuration. Therefore CRD/PRD profiles for these programs are stable across different architecture, and thus essentially architecture independent and can provide accurate analysis.

As multicore RD analysis is a great tool to study cache size scaling, there are also researches that extended reuse distance to study core count and problem size scaling. Researches shows that CRD and PRD profiles for symmetric threads change in a systematic fashion when scaling core count and problem size. For core count scaling, memory interleaving from the threads are increased, but with similar locality characteristics [29–31]. For problem scaling, the computation structure, such as loops, and the data structure are increased proportionally [30, 47]. Therefore the shift in CRD and PRD profiles preserve the shape and are highly predictable.

Therefore researchers proposed using simple prediction algorithms to predict the profiles with different core counts and problem sizes. Reference groups [47] was proposed to predict shape-preserving profile shift. The technique creates groups of reference and measure the shift rates for each group by comparing two profiles. Then apply the scaled shift rates to form a scaled-up profiles. The technique is first used to predict RD profiles across problem size scaling [47]. It is then extend to predict CRD and PRD profiles across problem size and core count scaling, and researches show the technique is very accurate [29, 30, 47].

3.2 Multicore Reuse Distance Profile

3.2.1 CRD Profile

As explained in Section 3.1, *Concurrent reuse distance* or CRD, is used to capture locality information in a shared cache, by measuring the reuse distance of thread-interleaved memory reference stream that access one shared cache. CRD values are measured on a global shared LRU stack [25–31].

To illustrate, Figure 3.1(b) shows a interleaving of memory references from two cores, C_1 and C_2. In this memory access stream, all memory references are read expect the reference to block D at $t = 10$, as indicated by (w) in the figure. This memory access stream is constructed from the memory access stream in Figure 3.1(a), by distributing and interleaving the memory accesses among two cores. Figure 3.2(b) and (c) shows the global LRU stacks for CRD profile. In particular, Figure 3.2(b) shows the state of the global LRU stack at $t = 9$, when core C_1 references block A. There are six blocks above A in this global LRU stack, thus the CRD for this access is 6, indicating this access is a hit for any shared cache with 7 or more entries, but a miss for any shared cache with less than 7 entries.

However, the RD for the accesses belong to core C_1, also know as intra-thread RD, is 3, while CRD for this access is 6. In this case, CRD > RD, because of the memory interleaving. The accesses from C_2 is different from accesses from C_1, and cause *dilation* of intra-thread reuse distance. Therefore the private data in the threads will cause CRD to increase compared to its intra-thread RD.

On the other hand, the shared data in the threads can offset the dilation effect. First effect is the *overlapping reference*. For example, Figure 3.1(b) shows C_2's access to block C at $t = 8$ interleaves with C_1's reuse of block A at $t = 9$, but it does not increase A's CRD, because C_1 already accesses block C at $t = 5$ in this reuse interval. Second effect is the *intercept*. To illustrate, Figure 3.2(c) shows the state of the global LRU stack at $t = 11$, when core C_1 is referencing block C. The intra-thread RD for this access is 2, while the CRD for this access is also 2. This is because core C_2 accesses the same block C in this reuse interval, causing the CRD to decrease.

3.2.2 PRD Profile

As explained in Section 3.1, *Private-stack reuse distance* or PRD, is used to capture locality information in private caches, by measuring reuse distance across per-thread memory reference streams that access coherent private caches. PRD values are measured on multiple private LRU stacks that are kept coherent with individual threads' memory reference streams [27–30].

Similar to sequential RD profiles, PRD profiles capture the reuse within each LRU stack, or intra-thread reuse, which can be used to predict the private cache misses. Figure 4.3(d)-(f) shows the LRU stacks for PRD profiles. In particular, Figure 4.3(d) shows the intra-thread reuse at $t = 9$. In Figure 4.3(d), Core C_1 references A. There are three blocks above A in the LRU stack of core C_1, so its PRD = 3. This access is a hit in a cache of size 4 or more, but a miss in a smaller

cache. But the private stacks are multiplied, so the total private cache size should be the aggregate capacity of all private cache. Therefore in this case, the private cache has to be size 8 or more for this access to be a hit.

Other than intra-thread reuse, PRD profiling captures inter-thread interactions as well, *e.g.*, sharing. When read sharing occurs, in which one data block is accessed by multiple cores; replicas of the block show up in multiple stacks. These replicas in the stacks increase the capacity pressure because more cache is needed to satisfy the cores' accesses. For example, in Figure 4.2, C_1 accesses c at $t = 5$ and C_2 accesses C at at $t = 8$. Figure 4.3(d) shows data block C is replicated in both stacks.

In addition to read sharing, PRD also captures write sharing by maintaining coherence via write invalidation between LRU stacks. To illustrate, C_2's reference to D at $t = 10$ in Figure 4.2 is a write , then invalidation will occur in C_1's stack as in Figure 4.3(e). The invalidation has two consequences. One is that a "hole" is created when a block is invalidated to prevent promotion of blocks further down the LRU stack [27]. When a data block below the hole is referenced, the hole moves to the position of that block, to preserve the stack depth of the blocks below. For example, in Figure 4.3(f) shows the state of the private LRU stack at $t = 12$, after the re-references of C by C_1 at $t = 11$. Comparing to Figure 4.3(e), Figure 4.3(f) shows, the reference of C cause block A to be pushed down and the hole move to depth (C's old position). The other consequence is the coherence miss, which is a miss caused by the write invalidation. For example, in Figure 4.3(f), after invalidation, if C_1 re-reference block D again, then this access will always miss, regardless of the

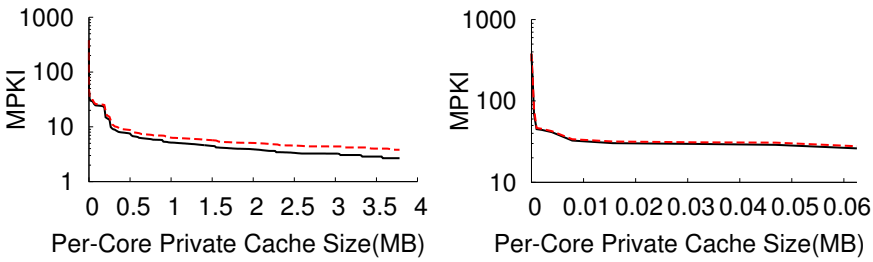
cache capacity.

3.3 Cache Sharing Behavior

In previous studies [29,30], researchers studies the sharing in the caches using reuse distance analysis, in particular the difference between CRD and PRD profiles. This thesis studies the directory behavior using an extension of reuse distance analysis. Since the sharing in cache is closely related to the directory behavior, this section reviews how CRD and PRD profiles reveal the sharing in caches.

As explained in Section 3.2, PRD profile and CRD profile are used to study private cache and shared cache respectively. If there is no sharing between the threads, the profiles for private cache and shared cache are similar, as shown in previous studies [29, 31]. But sharing will cause PRD profile to have more misses than CRD profile. There are two sources of the extra misses.

One is the extra misses due to the replication. To illustrate, Figure 4.3(d) shows the core C_1 re-referencing block A at $t = 9$. Figure 4.3(d) also shows data block C is replicated in both stacks and this replication puts pressure to the cache. And as explained in Section 3.2.2, for access A to be a hit, the private cache must be size 8 or more. But Figure 4.3(b) shows the same access for shared cache, and as explained above, for this access A to be a hit, the shared cache only need to be size 7 or more. The difference in capacity requirement comes from the replication of block C. Therefore, for same capacity, the private cache will suffer more misses than shared cache due to replications.



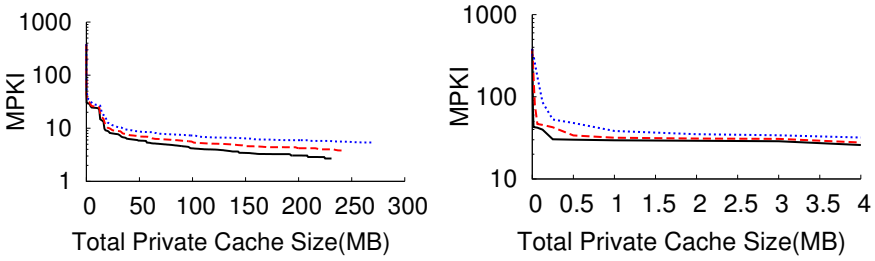
(1) ocean (2) ocean for small capacity

Figure 3.3: CRD and PRD profiles for benchmark ocean

The other source of extra misses is the coherence miss from the invalidations. Coherence miss can only happens in private cache. To illustrate, in Figure 4.3(f), after invalidation of block D at $t = 10$, if C_1 re-reference block D again, then this access will always miss, regardless of the cache capacity. While in Figure 4.3(c), the write to block D at $t = 10$ will not cause any invalidation. After the access of block C at $t = 11$, if C_1 re-reference block D, it will still hit if the cache is large enough. In another word, coherence miss will never happen in CRD profile because there is no write invalidation in a shared cache. Therefore, the private cache will suffer more cache misses than shared cache due to invalidations.

To illustrate the difference between CRD and PRD profiles, Figure 3.3 plots the MPKI corresponding to the CRD and PRD profiles for ocean from SPAHSH2 benchmark suites [48] following the method in the previous study [29, 30]. Figure 3.3(1) shows the two complete profiles and Figure 3.3(2) shows the two profiles for small capacity.

As Figure 3.3(2) shows, the PRD and CRD profiles are almost identical at small capacity, indicating the absence of sharing effects, both replications and invalidations. This shows at small capacity, the most data in the cache are private data.



(1) ocean (2) ocean for small capacity

Figure 3.4: PRD profiles for benchmark ocean for 16-, 64-, 256-cores

While figure 3.3(1) shows in larger capacity, there is a gap between PRD and CRD profiles, which is called "sharing-based degradation" in previous study [30]. This shows the sharing only begin to manifest itself in a larger cache.

This is because the parallelization scheme, such as blocking, usually have high sharing distance. Traditionally, programmers try to avoid frequency sharing in the benchmarks to minimize the communication and getting better performance. Therefore, the sharing distance for benchmarks are usually high and thus the sharing is only observable in a larger cache.

3.4 Core Count Scaling Behavior

In previous studies [29, 30], researches also study how PRD profiles changes with core count scaling. This thesis also studies how directory behavior changes with core count scaling. Therefore this section reviews the core count scaling effects on PRD profiles.

Figure 3.4 plots the MPKI corresponding to the PRD profiles for ocean from SPAHSH2 benchmark suites [48] for 16-, 64- and 256-cores, following the method in the previous study [29, 30]. Figure 3.4(1) shows the three complete profiles and

Figure 3.4(2) shows the three profiles for small capacity.

As Figure 3.4(2) shows, at small capacity, the PRD profile exhibits a shift with the increasing core count. In fact the shift is proportional to the number of cores. This is because the core count scaling increases the interference between the threads. In loop-based parallel programs, the parallelization is done by breaking the loop into multiple parts and execute them on multiple cores, as Figure 3.1 shows. Therefore, with more cores, more interleaved memory locations are referenced between reuses. And the increased PRD value shifts the PRD profiles towards higher capacity.

While as Figure 3.4(1) shows, at larger capacity, the shift in PRD profile becomes smaller and not proportional to the number of cores. This is because of the limited scope of the interference. In loop-based parallel programs, though the distance of reuse within a loop is increased due to more core count, the distance of reuse between the loops are mostly constant. Therefore, at larger core count the shift is very insignificant.

On the other hand, Figure 3.4(1) shows the non-shifting parts of PRD also increases with core count. This is due to the increase of sharing with core count scaling. With more cores, there are more replications and invalidations, causing PRD to increase with core counts.

Chapter 4: Analysis Framework

This chapter explains the how reuse distance framework is adapted for analyzing directory cache. Section 4.1 discusses how to study the directory behavior through the data cache transactions information. Section 4.2 and 4.3 explain using reuse distance analysis framework to study the directory access and content. Section 4.4 discusses the sensitivity of the analysis framework to coherence protocol.

4.1 Characterization of Directory Behavior

A directory cache access is performed when a core performs a memory operation that cannot be satisfied from the core's local private cache hierarchy, requiring memory transactions that involve other remote caches or main memory. Therefore the behavior of a directory cache can be studied through data cache transactions. To illustrate, Figure 4.1 groups cache transactions into 3 categories, labeled T1–T3.

T1 represents a cache transaction that misses all the way to the sharing point, either to shared cache or main memory. The directory is accessed, to create an entry to track future coherence information. Moreover, the directory access that caused by T1 transaction is usually latency tolerant, because T1 transaction incurs access below the sharing points, often off-chip, and much higher access latency than

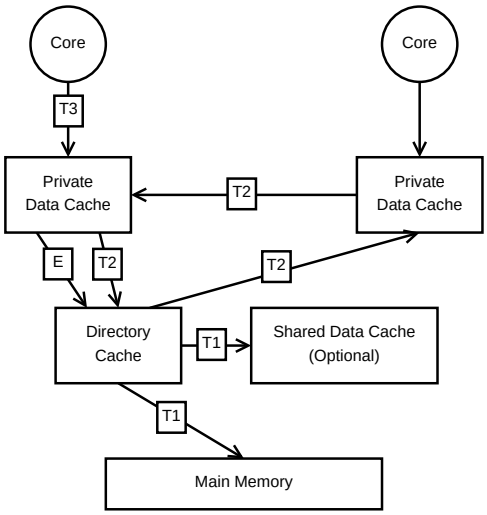


Figure 4.1: Directory Accesses in a Multicore Cache Hierarchy

directory access. Though directory still need be to accessed to determine that the data is not chip, it can be overlapped with the memory access because only memory address is needed to proceed.

T2 represents a cache transaction that requires information from remote cores. These "sharing-based" transaction accesses directory too, to determine the coherence action and the sharers involved. Comparing to T1 transaction, the directory access that caused by T2 transaction is usually latency critical, because the directory information is essential to either the correctness of the program or getting the data, the directory access has to done before getting the data block.

T3 represents a cache transaction that hit in local private data cache and can be satisfied by it, such as read hit or write hit to a modified block. T3 transaction will not generate directory access.

Finally, E represents a notification to the directory when when a cache block is evicted from the private cache. This thesis assumes all eviction, dirty or clean

will notify the directory, so that the directory is synchronized with the cache.

Therefore, by examining the data cache transactions and data cache eviction, one can get the access information to the directory cache. This includes the access frequency to the whole directory, as well as the how the access distributed across directory entries.

Apart from directory cache access frequency and distribution, the access to the directory also change the content of the directory. In order to analyze the directory cache content, the concept of *directory entry lifetime* is introduced. *Directory entry lifetime* is defined as the period that the entry resides in the directory.

When T1 transaction is performed, a new data block is brought into the private cache. This creates a new directory entry with a single sharer in the directory cache, and starts a new directory entry lifetime.

During its lifetime, T2 transaction access modifies the sharer list of the directory entry. A T2 read transaction adds a sharer to the entry's sharer list, while a T2 write transaction reduces the sharer list to a single sharer (assuming invalidation on writes). And a eviction notification also subtract one sharer from the sharer list.

Finally, the lifetime of a directory entry ends when the directory is notified that all copies of its associated data blocks have been evicted from the private caches. And the directory entry can be deallocated to provide space for other entries.

Therefore, by examining the data cache transactions and data cache eviction, one can get the content information to the directory cache. This includes total number of entries in the directory as well as the sharing degree of each entry.

Here, the T1-T3 and E transactions in Figure 4.1 are determined by the private

data cache. The data cache size affects the hit and miss of a data transaction and the number of private caches (*i.e.*, cores) will affect the sharing between each caches, thus affect the balance and frequency of T1, T2, T3 and E transactions. As discussed above, this not only changes the access pattern but also affects the content of the directory by changing the lifetimes and the sharer lists of a directory entry. The goal of this thesis is to fully characterize the dependence of directory's accesses and contents on the private data cache hierarchy.

4.2 Directory Access Analysis

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Core C_1:	A	B	C		D	E	A			B			C		B	C
Core C_2:				E				C	F		G	E		H		
								(w)								

Figure 4.2: Two interleaved memory reference streams

As explained in Chapter 3, RD analysis is a great tool to study the cache behavior with cache scaling, because one profile can provide the cache information across all cache size. This section discusses how to extend multicore RD analysis to analyze directory caches. As discussed in Section 4.1, the directory's accesses are closely related to data cache misses. In particular, a major part of the directory's accesses come from cache misses, because when a cache miss occurs, the directory is accessed to determine if the requested cache block is in other private caches.

Figure 4.2 shows a interleaving of memory references from two cores, C_1 and C_2. In this memory access stream, all memory references are read expect the reference to block C at $t = 8$, as indicated by (w) in the figure. For example, when

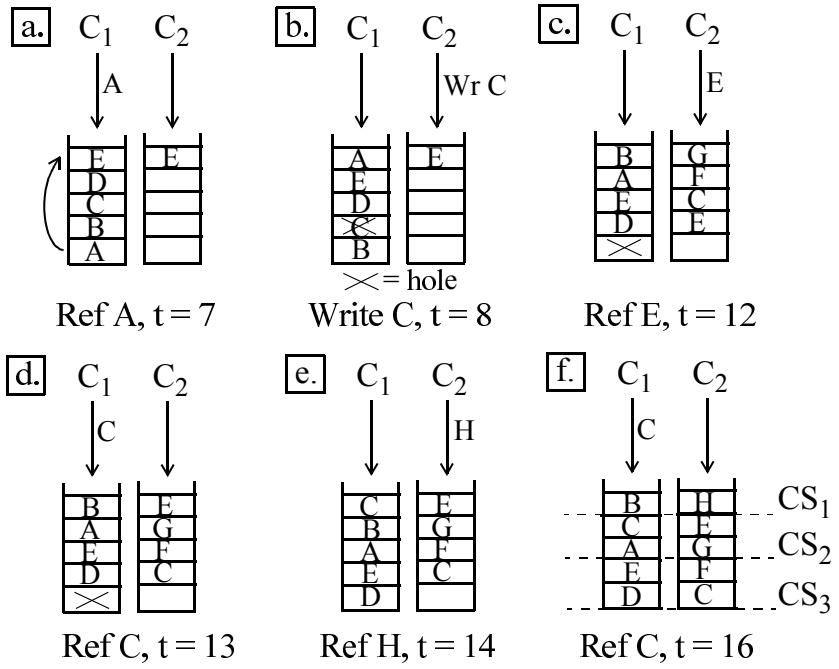


Figure 4.3: LRU stacks

C_1 re-references block A at $t = 7$ in Figure 4.3(a), if the local private cache is large enough to capture this reuse (PRD $< CS$), then the reference is a read hit. Otherwise (PRD $\geq CS$), the reference is a miss and generates a directory access. Therefore, looking at PRD already provides a major part of the directory access information.

However, PRD profile only provide local cache information, and this information alone is not enough. The problem is revealing the sharing in the cache. For example, PRD profiles cannot identify if a write hit is performed on a shared cache, nor can they distinguish whether a directory access is a shared access (T2). As discussed in Chapter 1, although sharing is an application-level property, whether the sharing in the program is exposed to the directory depends on the size of the data cache. Therefore, the *relative reuse distance* between sharers is essential to

determine if the sharing happens in the cache.

Relative reuse distance is proposed to evaluate the difference of stack depths for the same block across multiple stacks. Because the stacks are organized in LRU fashion, the difference in stack depths indicates how far apart in time the sharing happens. For example, if the two accesses to the same block from different cores happen close in time, then a small cache is sufficient to retain the data block when the second access happens. On the other hand, if the two accesses happen far apart in time, the data block is possibly evicted from the cache before the second access happens if the cache is too small. The sharing is only exposed given a bigger cache.

In order to analyze the relative reuse distance, *remote reuse distance*, or PRD_{remote}, is introduced. PRD_{remote} is the minimum stack depth across all remote PRD stacks for a memory reference. Relative reuse distance can be calculated by comparing PRD and PRD_{remote}, thus obtaining the sharing information.

There are three possible outcomes for $\text{PRD}_{remote} < CS$, $\geq CS$ and ∞. When $\text{PRD}_{remote} = \infty$, there is only one copy of the data block regardless of cache size, and it resides in the local LRU stack. Thus, this data block is "truly private." $\text{PRD}_{remote} \geq CS$ suggests there is no other copy of this data block within the limited time window captured by CS. The data block is classified as "temporally private" [22]. Lastly, $\text{PRD}_{remote} < CS$ shows there are other copies in the remote LRU stacks, and there is sharing going on in the private caches.

To illustrate, Figure 4.3(f) shows C_1 re-references data block C at $t = 16$. Data block C exists in both LRU stacks, but at different depths, hence the directory access behavior depends on the private cache size. Figure 4.3(f) shows three cases, labeled

CS_1–CS_3. If the cache size is CS_1, then PRD $\geq CS_1$ and $\text{PRD}_{remote} \geq CS_1$, so neither copy is on chip. Thus, C_1's reference is a miss and generates a T1 directory access. If the cache size is CS_2, then PRD $< CS_2$ and $\text{PRD}_{remote} \geq CS_2$, and only C_1's copy is on chip. The block C is "temporally private." So C_1's reference is a T3 access regardless of access mode, and does not generate a directory access. Lastly, if the cache size is CS_3, then PRD $< CS_3$ and $\text{PRD}_{remote} < CS_3$, and both copies are on-chip. If the access is a read, this is a read hit and a T3 access, so there is no access to the directory. But if the access is a write, then this is a write hit to a shared block, so a T2 directory access is generated.

The above example shows a few combinations of access mode, PRD and PRD_{remote}, and how they affect the access to the directory. Table 4.1 lists the 18 possible different cache transactions, depending on the access mode (read or write), PRD/PRD_{remote} outcomes discussed above, and also how these transactions are related to MESI protocol in Table 2.1. Table 4.1 reports them in terms of the T1–T3 categories.

The first eight transactions in Table 4.1 are from the T1 category. All of these accesses miss in the local private cache and there is no other copy in remote private caches (PRD $\geq CS$ and $\text{PRD}_{remote} \geq CS$). Transactions 1 and 2 represent cold misses, as shown in Figure 4.3(e). Transactions 2 and 3 can be a local cold miss, but in most cases they are coherence misses, which is an access after invalidation. As shown in Figure 4.3(d) when $CS < 4$, PRD for block C is ∞ because of the write invalidation that happens at $t = 8$. Transactions 5 and 6 represent the case where the data is truly private and resides only in the local cache, and corresponds to

	Mode	PRD	PRD_{remote}	sharer count	Comment	Table 2.1
T1 Transactions: New Lifetimes						
1	R	∞	∞	0 -> 1	Cold Miss	1
2	W	∞	∞	0 -> 1	Cold Miss	4
3	R	∞	$\geq CS$	0 -> 1	Coherence Miss	1
4	W	∞	$\geq CS$	0 -> 1	Coherence Miss	4
5	R	$\geq CS$	∞	0 -> 1	Truly Private	1
6	W	$\geq CS$	∞	0 -> 1	Truly Private	4
7	R	$\geq CS$	$\geq CS$	0 -> 1	Temporally Private	1
8	W	$\geq CS$	$\geq CS$	0 -> 1	Temporally Private	4
T2 Transactions: Directory Entry Reuse						
9	R	∞	$< CS$	N -> N+1	Forwarding, Coherence Miss	2
10	R	$\geq CS$	$< CS$	N -> N+1	Forwarding	2
11	W	∞	$< CS$	N -> 1	Invalidation, Coherence Miss	5
12	W	$\geq CS$	$< CS$	N -> 1	Invalidation	5
13	W	$< CS$	$< CS$	N -> 1	Invalidation	6
T3 Transactions: Data Cache Hits						
14	R	$< CS$	∞	1 -> 1	Truly Private	3
15	W	$< CS$	∞	1 -> 1	Truly Private	7
16	R	$< CS$	$\geq CS$	1 -> 1	Temporally Private	3
17	W	$< CS$	$\geq CS$	1 -> 1	Temporally Private	7
18	R	$< CS$	$< CS$	N -> N	Read to Shared	3

Table 4.1: Access mode, PRD, PRD_{remote} and sharer count characterization of data cache transactions and T1–T3 categorization.

Figure 4.3(a) when $CS < 5$. And transactions 7 and 8 represent temporally private data, as the sharing is not captured on chip, and corresponds to Figure 4.3(f) when $CS = CS_1$. All the reads in this categories correspond to case 1 in Table 2.1 for MESI protocol, indicating the directory block transitioned from invalid state to exclusive state. While all the writes in this categories correspond to case 4, indicating the directory block transitioned from invalid state to modified state.

The next five transactions in Table 4.1 are from the T2 category. All of these accesses involve sharing that is captured on-chip ($\text{PRD}_{remote} < CS$) and require remote action such as invalidation or forwarding. Transactions 9 and 10 represent a read miss in the local private cache, but the data can be forwarded by a

remote private cache. Transaction 9 corresponds to Figure 4.3(d) when $CS \geq 4$ and transaction 10 corresponds to Figure 4.3(c) when $CS = 3$. They also correspond to case 2 in Table 2.1 for MESI protocol, indicating the directory block transitioned to shared state. Transactions 11 and 12 represent a write miss to a shared block on chip, which causes invalidation. Transaction 10 corresponds to Figure 4.3(b) when $CS \geq 4$ and transaction 11 corresponds to Figure 4.3(c) when $CS = 3$. They also correspond to case 5 in Table 2.1 for MESI protocol, indicating the directory block transitioned to modified state. Transaction 13 represents a write hit to a shared block on chip, also causes invalidation. It corresponds to Figure 4.3(f) when $CS = CS_3$. It also corresponds to case 6 in Table 2.1 for MESI protocol, indicating the directory block transitioned to modified state. Also, similar to transactions 2 and 3, transactions 9 and 11 can be a local cold miss, but in most cases they are coherence misses.

The last five transactions in Table 4.1 are from the T3 category. All of these accesses hit in the local private cache (PRD $< CS$) and there is no need for a coherence operation. Transactions 14 and 15 represent accesses to truly private data, and correspond to Figure 4.3(a) when $CS \geq 5$. Transactions 16 and 17 represent accesses to temporally private data, and correspond to Figure 4.3(f) when $CS = CS_2$. Transactions 18 represents a read to a shared block, as shown in Figure 4.3(f) when $CS = CS_3$ and the reference is a read. All the reads in this categories are corresponding to case 3 in Table 2.1 for MESI protocol, indicating no change in directory states, while all the writes are corresponding to case 7, indicating the directory block transitioned to modified state locally.

4.2.1 Evictions

In addition to T1 and T2 transactions, evictions from data caches also access the directory, (*i.e.*, to notify the directory that the block is no longer in the data cache). Evictions can also be identified by PRD profiling. When a block is accessed, it pushes the blocks in the local LRU stack downward. And when a block moves below a given stack depth, it is evicted from the cache with the corresponding capacity. For example, in Figure 4.3(c), block E is accessed, and as shown in Figure 4.3(d), block G, F and C are pushed down. Therefore, if $CS = 1$, block G is evicted, if $CS = 2$, block F is evicted, while if $CS = 3$, block C is evicted.

4.3 Directory Content Analysis

Besides directory accesses, another important part of the directory behavior is the contents of the directory. Table 4.1 also reports how different cache transactions change the directory's contents.

To analyze the content of the directory, this thesis first studies the number of live entries in the directory. As discussed in Section 4.1, a T1 transaction in Table 4.1 represents a cache transaction that misses all the way to the sharing point, introducing a new entry into the directory for future use. Hence, this transaction increases the number of entries by one. On the other hand, during a data block eviction, the directory is notified. When the last copy of the cache block is evicted, indicating there are no copies in the private caches, the directory entry is evicted and the number of directory entries decreases by one.

This thesis also further studies the content of the directory by analyzing the sharer count of each directory entry. When a directory entry is first introduced by a T1 transaction, the sharer count of the entry is one, meaning there is only one copy of this cache block in the private cache. While this entry is in the directory, it might receive T2 transactions. T2 transactions in Table 4.1 represent sharing-based directory transactions. These will change the sharer count of the directory entry. Read transactions (transactions 9 and 10 in Table 4.1) will increase the sharing count by one, and write transactions (transactions 11, 12 and 13 in Table 4.1) will reduce the sharing count to one through invalidations. On a data block eviction, the directory is notified which decreases the sharer count for the directory entry by one. Eventually, the sharer count will reach zero when there are no copies in the private caches. This coincides with the eviction of the directory entry.

4.4 Sensitivity to Coherence Protocols

Although this work focuses on analyzing the access patterns and contents of the directory across many architecture configurations, it is inevitably sensitive to coherence protocols. For coherence protocols differ in the cache statues they support.

This thesis assumes the MESI protocol which includes the exclusive state. But as explained in Section 2.1.2, depending on the protocol, some cache transactions may or may not become a directory access. For example, the MSI protocol does not have exclusive state. A write hit to a clean block will result in a directory access regardless if the block is private or not, while in the MESI protocol a write hit to a

clean but private block will not result in a directory access. Therefore a directory using the MSI protocol incurs more accesses to the directory.

Another example is MOESI protocol. MOESI protocol have owned state, which is a dirty cache block shared in the private cache. Comparing to shared state in MESI protocol, the data in owned state in the private cache is inconstant with its copy in the next level. This affect the latency criticality of the access to this directory entry. For example, a read miss to a block in shared state can access the directory and next level memory in parallel because the next level have the most recent information. But a read miss to a block in owned state must access the directory first before getting the data, because the directory holds the location of the most recent data.

On the other hand, the eviction notification policy affects the directory content information. This thesis assumes all evictions notify the directory, so that the directory has full knowledge of the content of the cache. Many directory techniques adopted this policy, such as Cuckoo Directory [19], DGD [20] and SCD [18]. But some directory techniques does not inform directory after a clean data cache evictions, so that it will consume less memory bandwidth, as in [18–20, 22].

First, when directory is not notified when a clean data evicted from cache, the directory entry will record extra sharers that does not have the data any more, thus the sharing degree of the directory entry can be inaccurate. For example, assuming the data is cached by cache 1 and 2, then get evicted by cache 1 but cached by cache 3, the sharer list of the directory that gets clean notification will be 2,3 while the sharer list of the directory that doesn't get clean notification will be 1,2,3.

Second, when directory is not notified when a clean data evicted from cache, the directory cannot evict the directory entry when its lifetime is ended. For example, assuming the data is cached by cache 1 and 2, then get evicted by both caches, the directory can not evict this entry because it does not get the eviction notification. Therefore in this case, the directory contents is different from this analysis because some entries stay in directory even if their lifetimes have ended.

Chapter 5: Directory Cache Profiler

5.1 Profiler Process Flowchart

This thesis implements the directory behavior profiling presented in Chapter 4 within the Intel PIN tool [32]. Figure 5.1 shows the flowchart of the whole process. The Intel PIN tool is able to instrument memory instructions in the binary. For every memory instruction, the Intel PIN tool passes the memory address, core id and the read/write information to the PIN profiler.

As discussed in Section 3.2, the PIN profiler maintains coherent private LRU stacks. (64-byte blocks are assumed in all LRU stacks). For every memory reference, the PIN profiler first consults the LRU stacks to compute PRD and PRD_{remote}. Then it refers to Table 4.1 to determine the directory access type. In order to enable capacity scaling analysis, the PIN profiler refers to the table multiple times

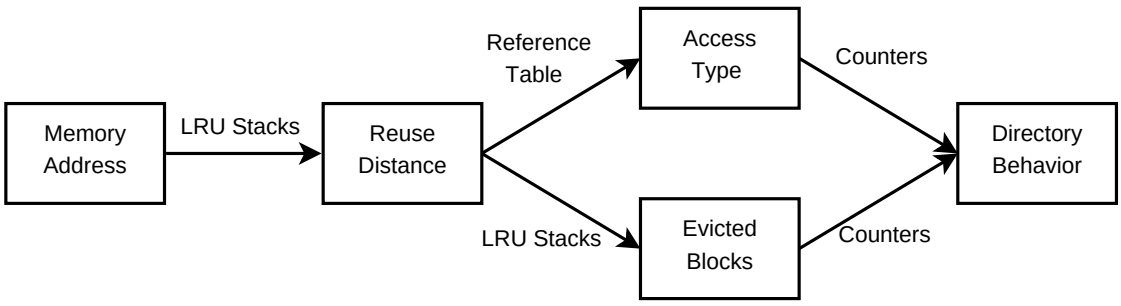


Figure 5.1: Process Flowchart of the Profiler

for different CS values. While the LRU stacks can explore all CS exhaustively, the PIN profiler steps CS in increments of 16KB and stop at the application's maximum PRD for profiling speed.

Since computing PRD and PRD_{remote} requires examining all LRU stacks, both local and remote, other information can also be obtained at the same time besides the cache transaction and directory access type. One additional piece of information acquired is evicted blocks for each capacity CS. This can be obtained by searching the block that is pushed to stack depth CS in the current local LRU stack. Another piece of information is the current number of sharers for each capacity CS. This can be obtained by computing the number of cores with stack depth less than CS.

After getting the directory access type and the evicted block for each capacity CS, the PIN profiler updates a set of counters, for both directory access information and directory content information. Finally, based on the statistics the PIN profiler collects, the behavior of the directory is obtained

Finally, the PIN profiler follows McCurdy's method [49] which performs functional execution only, context switching threads after every memory reference. This interleaves threads' memory references uniformly in time. Studies have shown that for parallel programs with symmetric threads, this approach yields profiles that accurately reflect locality on real CPUs [26, 29], especially for PRD profiles.

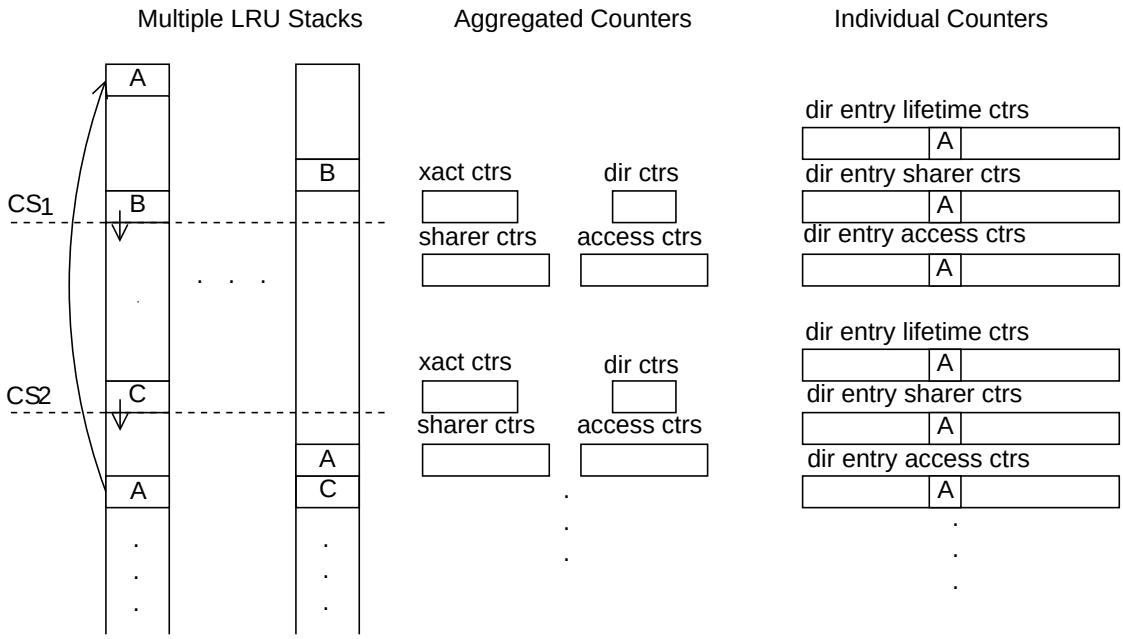


Figure 5.2: PIN profiler implementation.

5.2 Profiler Implementation

This section explains the implementation of the PIN profiler. Figure 5.2 illustrates the two parts of the PIN profiler. One part is the LRU stacks, which provide directory information based on memory accesses, such as PRD and PRD_{remote}. The other part is the counters, which record different statistics that the profiler tracks.

5.2.1 LRU Stacks

As discussed in 3.2, the PIN profiler uses LRU stacks to obtain the PRD and PRD_{remote} value for each memory reference. LRU stacks are implemented using top-down splaying tree [50]. This binary search tree is sorted by access time, and therefore maintains the LRU ordering of blocks in the cache. Also, the rank of a node is defined as the number of nodes which have higher access time in the tree.

When an access is made to block A in the LRU stack, the profiler uses a hash table to find the last access time for the block A, and searches the tree. Then the depth of block A is the rank of the node A. Therefore, by examining all the stacks, PRD and PRD_{remote} for this access can be obtained.

It is also possible to get the address for evicted blocks in the LRU stacks. When an access is made to block A in the LRU stack, it will update its place in the stack and will increase the rank of other blocks in the stacks. So the block that gets evicted from the cache with capacity CS is the block whose stack depth increases to CS. Therefore, by searching the blocks for the one whose rank increases to capacity CS, it is possible to find the evicted block for each capacity in the stack that is accessed.

5.2.2 Counters

The profiler maintains multiple sets of counters to keep statistics. Some are aggregated counters used to to keep the statistics for the whole directory, such as the access count to the directory and the average number of entries in the directory. Others are individual counters used to keep the statistics for individual directory entries, such as the access count to each entry and the sharer count of each entry.

First of all, the profiler records the access count to the directory to enable access frequency profiling. 19 aggregated counters are maintained for each CS value, one for each of the 18 cache transactions in Table 4.1 plus one for evictions. Figure 5.2 illustrates the per-transaction counters, labeled "xact ctrs," at each profiled

private cache size, labeled "CS_i." The profiler increments the corresponding cache transactions counter based on the result of the table look-up, and increments the eviction counter when evictions happen. To illustrate, Figure 5.2 shows a reference to block A. This reference is transaction 7 for capacities CS_1 and CS_2 by referring to Table 4.1, and also this transaction is a directory fill. Therefore, using these statistics and the total instruction count, it is possible to compute the frequency of directory fills, directory entry reuses and eviction notifications to the directory for all data cache capacities.

Second, the profiler tracks the number of live directory entries to enable the directory cache content profile. It computes the average number of live directory entries across time by accumulating the lifetime of all live directory entries and then averaging them by total time. In this study, the time is defined as the memory reference count. A set of directory lifetime counters are maintained, *one per unique data block contained in all of the LRU stacks for every capacity*, as shown in Figure 5.2, labeled as "dir entry lifetime ctrs." When a reference initiates a new directory entry lifetime, its time is stored into the counter. Figure 5.2 illustrates a reference to block A is a directory fill for capacities CS_1 and CS_2. And as discussed in Section 4.2, Table 4.1 shows this initiates a new directory entry lifetime for capacities CS_1 and CS_2. On the other hand, when an eviction terminates a directory entry, the profiler computes the difference between the stored time and current time as the duration of the lifetime of the entry. Figure 5.2 also shows block B is evicted at capacity CS_1, and block C is evicted at capacity CS_2. By consulting all LRU stacks when an eviction happens, the profiler determines that all copies of block C are at stack

depth greater or equal to CS_2. This indicates the lifetime of block C in the directory terminated, and the profiler computes the lifetime of block C at capacity CS_2 Then it adds this value into an aggregated counter for capacity CS_2. There is one aggregated counter provided for each capacity, labeled as "dir ctrs" in Figure 5.2. At the end of the program, these values are divided by the total time to obtain the average number of directory entries for each capacity.

In addition, the profiler obtains the sharing distribution in the directory by tracking the max sharing degree of individual directory entries during their time in the directory. Similar to directory cache content profiling, it computes the average number of live directory entries with N max sharing degree across time. A set of directory sharing counters are maintained along with the lifetime counters, one per data block for every capacity. Figure 5.2 illustrates these counters, labeled as "dir sharer ctrs." As mentioned above, when a reference is made, the profiler searches the LRU stacks to determine the current number of sharers for all profiled private cache sizes, and the counters are updated accordingly. When a directory entry is evicted at CS_i, the lifetime is added into an aggregated counter for capacity CS_i and N sharers. There is one aggregated counter provided for each capacity and each possible max sharing degree, labeled as "sharer ctrs" in Figure 5.2. At the end of the program, the profiler divides these values by the total time to break down the average number of directory entries in terms of the maximum number of sharers for each capacity.

Moreover, the profiler also counts accesses to individual directory entries during their lifetimes to enable the access distribution profile. In a similar fashion,

another set of per-entry counters at every CS is maintained, labeled "dir entry access ctrs" in Figure 5.2. Each time a reference is made, the profiler checks if the transaction causes a directory access at capacity CS. If so, it increments the corresponding "dir entry access ctrs." Also, when a directory entry is evicted at CS_i, the profiler adds its lifetime into an aggregated counter for capacity CS_i and N accesses. There is one aggregated counter provided for each capacity and each possible access counts, labeled as "access ctrs" in Figure 5.2. At the end of the program, the profiler divides these values by time to break down the average number of directory entries in terms of the number of accesses received for each capacity.

Chapter 6: Profile Studies and Results

6.1 Experimental Setup

This chapter studies the directory's characteristics using the profiler and how multicore CPU scaling impacts the directory using 15 parallel benchmarks. Table 6.1 lists the benchmarks and their suites: SPLASH2 [48], MineBench [51], or PARSEC [52]. The last two columns in Table 6.1 report the problem sizes and their dynamic instruction counts (in billions). For the kernels, entire benchmark run is profiled. For all other benchmarks, the first parallel iteration is used to warm up the PRD stacks, and then the second parallel iteration is profiled.

Three studies are performed with profiler. Section 6.2 and 6.3 study how core count and cache size scaling affect the directory access stream and directory contents. Then Section 6.4 studies the directory access distribution across cache size scaling to show the temporal reuse of directory entries.

Benchmark	Suite	Problem Size	Instructions (Billions)
fft (kernel)	SPLASH2	2^{22} elements	2.42
lu (kernel)	SPLASH2	2048^2 elements	22.2
radix (kernel)	SPLASH2	2^{24} keys	3.79
barnes	SPLASH2	2^{19} particles	32.7
fmm	SPLASH2	2^{19} particles	16.4
ocean	SPLASH2	1026^2 grid	1.34
water	SPLASH2	40^3 molecules	2.31
kmeans	MineBench	2^{22} objects, 18 features	10.2
blackscholes	PARSEC	2^{22} options	2.44
bodytrack	PARSEC	B_261,16k particles	10.3
canneal	PARSEC	2500000.net	0.09
fluidanimate	PARSEC	in_500k.fluid	2.83
raytrace	PARSEC	1920x1080 pixels	4.22
swaptions	PARSEC	2^{18} swaptions	22.4
streamcluster	PARSEC	2^{18} data points	4.33

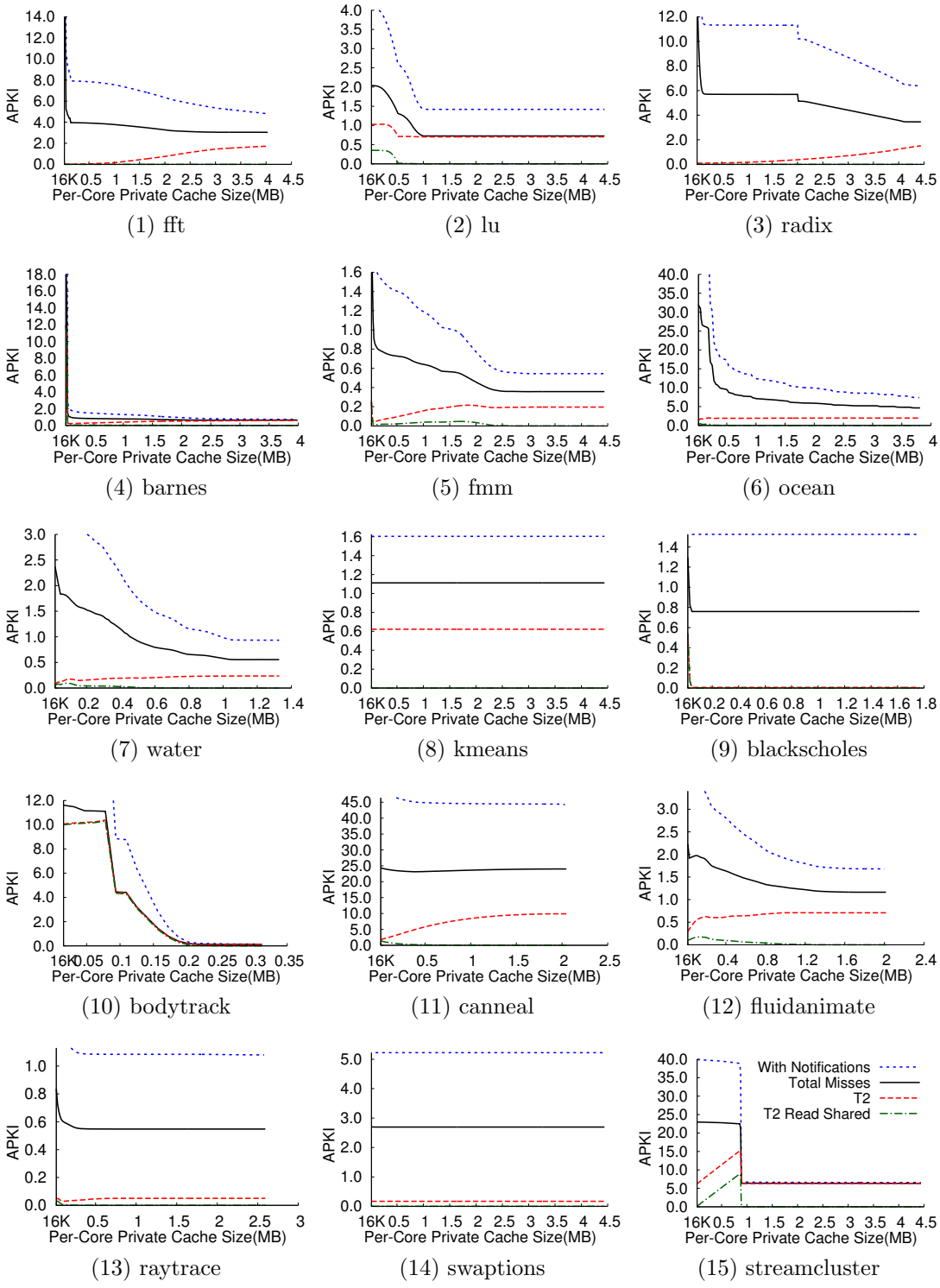
Table 6.1: Parallel benchmarks used in the evaluations.

6.2 Study 1: Directory Access Frequency

6.2.1 Cache Size Scaling

Figure 6.1 shows how scaling private data cache size can impact the cache-induced directory access frequency, as reported by the "xact ctrs" in the profiler. Figure 6.1 plots the total number of cache miss-induced directory accesses per 1000 instructions, or "APKI," in solid lines labeled as "Total Misses".

As shown in Figure 6.1, directory accesses are highly sensitive to data cache size, *they drop rapidly as capacity increases.* To illustrate, the first three columns in Table 6.2 show the access counts for all benchmarks at three different cache capacities. At small private cache sizes (16KB), 7 out of 15 benchmarks in Table 6.2 have a directory APKI exceeding 11, and one reaches 32. But at larger private cache



(1) fft (2) lu (3) radix

(4) barnes (5) fmm (6) ocean

(7) water (8) kmeans (9) blackscholes

(10) bodytrack (11) canneal (12) fluidanimate

(13) raytrace (14) swaptions (15) streamcluster

Figure 6.1: Breakdown of directory APKI *vs.* private data cache size for 64-core CPUs

Table 6.2: Cache-miss APKI at 3 private cache sizes, sharing-induced APKI, and APKI for 16- and 256-core CPUs.

Benchmark	Directory Accesses			T2			APKI	
	16KB	1MB	∞	16KB	1MB	∞	16c	256c
fft	16.0	3.8	3.0	0.0	0.2	1.7	3.7	3.9
lu	2.0	0.7	0.7	1.0	0.7	0.7	0.2	1.1
radix	16.7	5.7	3.4	0.1	0.2	2.3	5.6	6.1
barnes	19.1	0.8	0.6	12.0	0.4	0.6	0.6	0.9
fmm	2.0	0.6	0.4	0.3	0.2	0.2	0.6	0.8
ocean	32.0	7.1	4.6	1.3	1.9	2.0	6.0	10.1
water	2.4	0.6	0.6	0.1	0.2	0.2	0.5	0.8
kmeans	1.1	1.1	1.1	0.6	0.6	0.6	1.1	0.6
blackscholes	1.3	0.8	0.8	0.5	0.0	0.0	0.8	0.8
bodytrack	11.6	0.1	0.1	10.1	0.1	0.1	0.1	0.3
canneal	24.3	23.6	24.0	1.8	8.4	9.9	22.9	24.9
fluidanimate	2.2	1.3	1.2	0.3	0.7	0.7	0.8	1.9
raytrace	0.8	0.5	0.5	0.0	0.1	0.1	0.5	0.6
swaptions	2.7	2.7	2.7	0.2	0.2	0.2	2.6	2.9
streamcluster	23.0	6.4	6.4	6.2	6.3	6.3	5.7	6.9
Average	5.3	1.5	1.4	0.5	0.3	0.5	1.3	1.8

sizes (1MB), all benchmarks except for canneal have a directory APKI less than or equal to 7.1. And half of the benchmarks are under 1 APKI. Across all benchmarks, the average directory APKI drops from 5.3 at 16KB to 1.5 at 1MB, a factor of 3.5x.

The accesses to the directory can be broken down into two categories. One is the pure data cache capacity miss (T1 transactions), and the other is the on-chip sharing access (T2 transactions). To illustrate, Figure 6.1 plots APKI for T2 accesses in dashed lines, labeled as "T2." Therefore, the gap between the solid lines ("Total Misses") and the dashed lines ("T2") is the T1 accesses. Because the T1 accesses in Figure 6.1 are induced by data cache capacity misses, they decrease when the private cache size increases. The changes in T2 accesses with cache size scaling are more complicated, as discussed later. Moreover, the ratio between T1 accesses

and T2 accesses is also affected by the cache size scaling.

At small cache sizes, Figure 6.1 shows the cache accesses are mostly dominated by T1 transactions. The fourth to sixth column in Table 6.2 shows the T2 access counts for all benchmarks at three different cache capacities. They show that at small private cache sizes (16KB), T2 accesses are very few– most benchmarks (12 out of 15) have less than 2 APKI T2 transactions. By comparing the T2 access counts to total accesses counts in Table 6.2, it is shown in small private cache sizes (16KB), only 9% of the directory accesses are T2 accesses on average. This is because at 16 KB, the data caches are too small to capture many shared accesses occurring between threads, and data cache capacity misses are also high due to the small size of the cache. Therefore, the majority of the directory accesses are destined to private data, without incurring any sharing-based transactions. These data are also often temporally private because of the small cache size. There are three exceptions: lu, bodytrack and streamcluster. These benchmarks exhibit widely shared data even in a small data cache.

When data cache sizes increase, the ratio of T2 transactions over total cache-miss induced directory accesses generally increases too. By comparing column two and column five in Table 6.2, it is shown that in large private caches (1M), 20% of the directory accesses are T2 transactions. While T1 transactions decrease with cache capacity because data, especially truly private data, start to fit in cache and cease to incur T1 transactions, the reduction in T2 transactions is not as much, and even increases in 7 benchmarks. This is because as data starts fitting in the data cache, once temporally private data are no longer private, and sharing starts

to manifest on chip.

While the sharing is increasing with cache size scaling, depending on the type of sharing, it may cause an increase or decrease in the number of T2 transactions In particular, read sharing will cause an increase and then a decrease in T2 transactions. To illustrate, Figure 6.1 plots APKI for T2 accesses associated with read sharing (*i.e.*, transaction 10 in Table 4.1) in dash-dotted lines, labeled as "T2 Read Shared." "T2 Read Shared" transactions first increase with cache capacity because more remote sharers are captured on chip with bigger cache capacity. But once all sharers are cached, *i.e.*, the read-sharing working set fits in cache, there are no more directory accesses. In contrast, write sharing leads to coherence related T2 transactions, such as coherence misses(*i.e.*, transactions 9 and 11 in Table 4.1) and invalidations (*i.e.*, transactions 11, 12 and 13 in Table 4.1). These transactions also increase with capacity scaling because more sharing is captured with cache capacity scaling, but they cannot be eliminated by capturing all sharers on-chip. Therefore, the gap between the "T2" and "T2 Read Shared" curves in Figure 6.1 increases monotonically.

Moreover, at each benchmark's maximum PRD, which is equivalent to an infinite cache, all read-shared T2 transactions are eliminated while all write sharing is exposed. The sixth column in Table 6.2 shows T2 transactions reach 0.5 APKI at this maximum PRD. These T2 transactions at "∞" quantify a program's intrinsic coherence-related directory accesses. At maximum PRD, 36% of the directory accesses are T2 transactions.

Therefore, there are two effects of cache scaling in the directory. It decreases

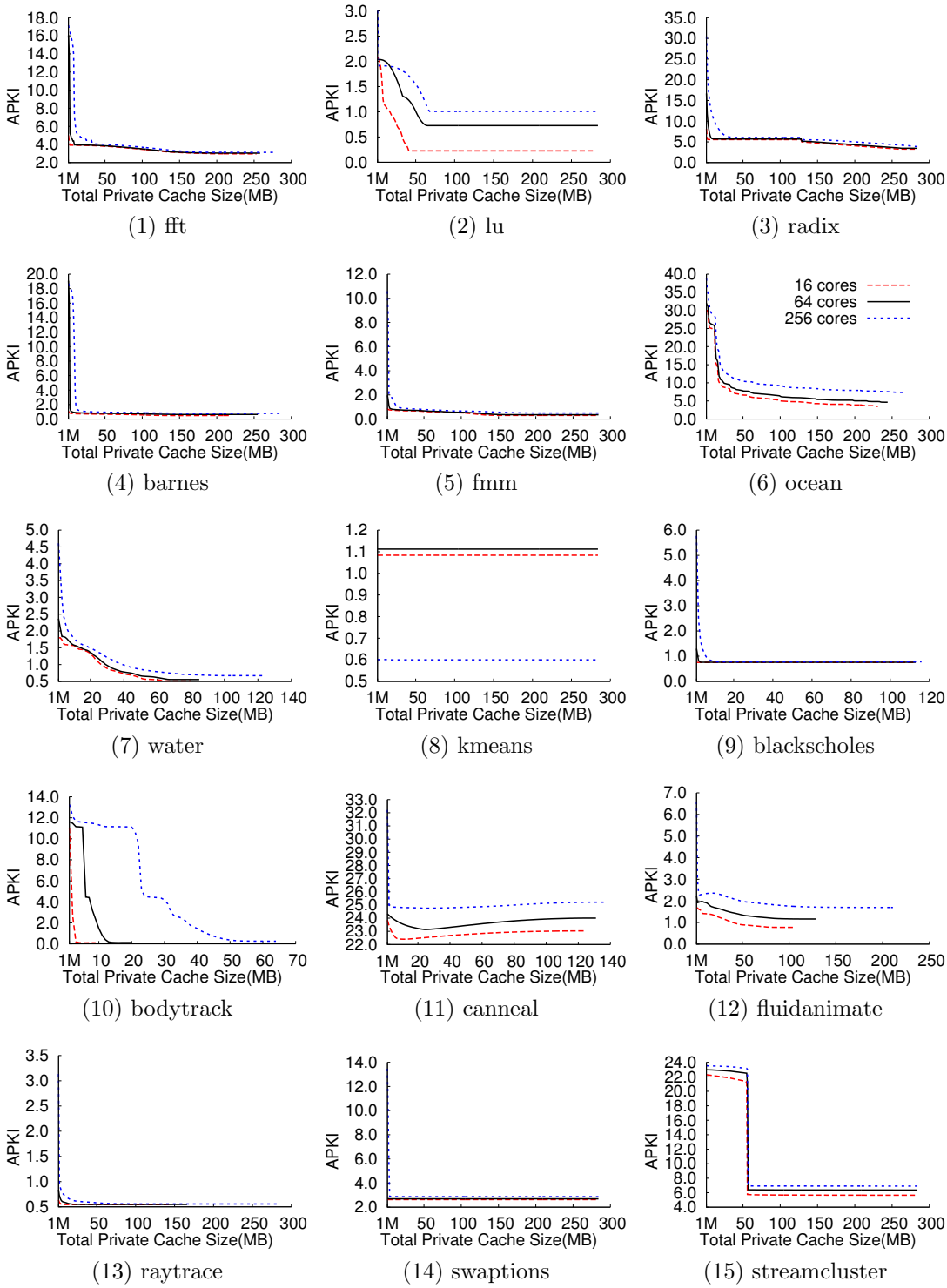
the T1 accesses because it decreases the cache capacity misses, especially to truly private data. It also increases the T2 accesses by exposing the sharing-based directory accesses on chip. What the results show is that the capacity effect has a bigger impact on the directory than the sharing effect. So the accesses to the directory decrease dramatically with cache size scaling.

6.2.2 Core Count Scaling

In addition to data cache scaling, core count scaling also affects the directory accesses. Figure 6.2 plots the total cache-miss induced directory APKI for three different core counts, 16, 64 and 256. Notice the x axis in this figure plots total cache size instead of per-core cache size as in Figure 6.1 to permit comparison across different number of cores.

As shown in Figure 6.1, core count scaling has a much smaller impact on directory access count than cache size scaling. To illustrate, the last two columns in Table 6.2 report the cache-miss induced directory accesses for 16- and 256-core CPUs at 64MB of total private cache. As shown in Table 6.2, average directory cache accesses only increase from 1.3 to 1.8 across all 15 benchmarks–a 38% increase–despite a 16x scaling in core count. In contrast, the previous section showed directory accesses decrease by 3.5x when scaling the private cache size from 16KB to 1MB.

As explained in the previous section, the directory accesses are mainly composed of cache misses, which follow the PRD profile. Previous studies have examined the effects of core count scaling on PRD [29, 30]. As explained in Section 3.4, the



(1) fft (2) lu (3) radix

(4) barnes (5) fmm (6) ocean

(7) water (8) kmeans (9) blackscholes

(10) bodytrack (11) canneal (12) fluidanimate

(13) raytrace (14) swaptions (15) streamcluster

Figure 6.2: Total cache-miss induced directory APKI for 16-, 64-. 256-core CPUs

main effect is that the PRD profile shifts to larger RD values as the number of cores increases, at small cache sizes. And then the PRD profile's shift slows down and becomes minimal for very large cache sizes. Therefore, because directory accesses are derived from the private cache misses that PRD profiles capture, the directory access profiles also exhibit the same shape-preserving shift.

Also, in most benchmarks, the total directory accesses increase slightly with the shifting. This is because of the increase in sharing-related accesses with the core count scaling. Therefore, the directory access increases with core count scaling, due to the curve-shifting caused by cache capacity misses and the increase in sharing accesses caused by increased number of cores. But the core count scaling has a much smaller impact on the directory access pattern compared to cache size scaling.

6.3 Study 2: Directory Coverage

6.3.1 Cache Size Scaling

Figure 6.3 shows how scaling private data cache size impacts the number of directory entries in the directory cache, as tracked by the "entry ctrs" in the profiler. This thesis uses a metric called *Coverage* [18], which is the ratio of the total live directory entries to total private cache blocks. Intuitively, if all the cache blocks in the private caches are privately accessed, there will be an equal number of directory entries and data cache blocks. So, the coverage is 100% in this case. If there are cache blocks that are shared among multiple cores, then some cache blocks will be replicated in the private cache, but only consume one entry in the directory. So the

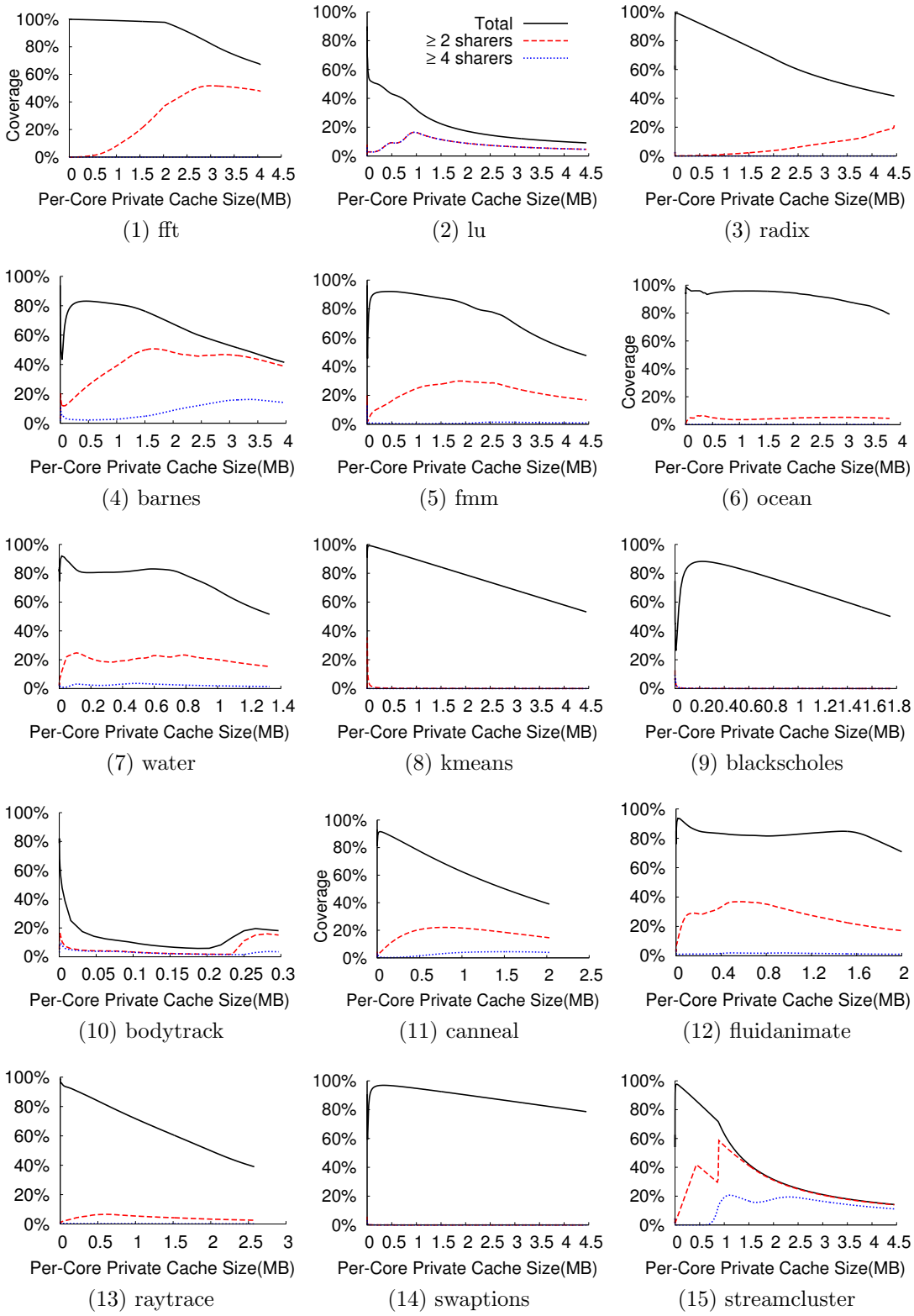


Figure 6.3: Coverage *vs.* private data cache size for 64-core CPUs

Table 6.3: Coverage for all entries

Benchmark	Coverage		
	64KB	1MB	∞
fft	99.9%	99.0%	67.1%
lu	52.3%	32.1%	9.1%
radix	98.8%	84.0%	41.6%
barnes	60.2%	81.0%	41.5%
fmm	85.8%	90.2%	47.6%
ocean	96.8%	95.8%	79.1%
water-spatial	87.5%	68.4%	51.6%
kmeans	99.1%	89.5%	53.2%
blackscholes	76.7%	70.9%	50.0%
bodytrack	12.4%	18.1%	18.1%
canneal	91.2%	62.5%	39.0%
fluidanimate	91.5%	82.3%	70.8%
raytrace	93.7%	71.7%	39.0%
swaptions	91.3%	94.8%	78.6%
streamcluster	97.2%	63.1%	14.2%
Average	82.3%	73.6%	46.7%

coverage less than 100% in this case. Figure 6.1 plots the time average result in solid lines, labeled as "Total". The results here are for 64-core CPUs.

As Figure 6.3 shows, *coverage decreases significantly with cache size scaling.* The coverage starts near 100% in most benchmarks in Figure 6.3, but then drops to near 50% in many benchmarks as the cache sizes increase. To illustrate, Table 6.3 reports coverage for each benchmark. At 64KB, the coverage is on average 82.3%, with 9 benchmarks over 90%. At 1MB, the coverage on average drops to 73.6%, with only 4 benchmarks over 90%. While at the maximum PRD, the coverage in half of the benchmarks drops below 50%, and reaches 46.7% on average. In extreme cases such as lu, bodytrack, and streamcluster, the coverage drops under 20%.

This drop in coverage is because of the increased sharing in larger caches. As discussed in Section 6.2, shared accesses between threads only start to be captured

in larger cache sizes, which leads to the increasing percentage of T2 accesses with data cache size scaling. Moreover, many of these T2 accesses increase the sharers tracked per directory entry as shown in Table 4.1. In other words, *the percentage of multi-sharer entries increases with cache size scaling, while the single-sharer entries decreases*. As explained above, because compared to private data blocks, shared data blocks can be tracked with fewer directory entries, the directory coverage drops with data cache size scaling.

To illustrate, Figure 6.3 plots the coverage for entries with 2 or more sharers in dashed lines, labeled as "≥2 sharers". So, the gap between the solid lines and dashed lines breaks down the coverage for single-sharer entries. As shown in Figure 6.3, the gap gets smaller with increasing data cache sizes. To quantify this phenomenon, the first three columns in Table 6.4 report the percentage of live directory entries that are multi-sharer entries. For 64KB private caches, only 9.33% of the directory entries are multi-sharer entries on average across all benchmarks, indicating over 90% of the entries serve private blocks in the cache. 9 out of 15 benchmarks have less than 6% shared entries and 4 benchmarks have less than 1% shared entries. At 1MB however, 27.99% are multi-sharer entries on average. The percentage of shared entries are more then doubled in 9 out of 15 benchmarks. And at the maximum PRD, 39.15% are multi-sharer entries.

However, further breaking down the coverage by number of sharers shows the increase in the sharing occurs non-uniformly. To illustrate, Figure 6.3 plots the coverage for entries with 4 or more sharers in dotted lines, labeled as "≥4 sharers." As in Figure 6.3, the gap between the dash lines and the dotted lines is big, showing

Table 6.4: Percentage of multi-shared entries

Benchmark	≥2 Sharers			≥4 Sharers		≥32 Sharers	
	64KB	1MB	∞	1MB	∞	1MB	∞
fft	0.01%	8.10%	71.09%	0.0008%	0.0188%	0.0008%	0.0184%
lu	5.54%	51.04%	50.92%	50.999%	50.890%	46.490%	47.325%
radix	0.16%	1.31%	51.19%	0.0207%	0.0400%	0.0014%	0.0022%
barnes	19.77%	48.24%	93.47%	3.3819%	34.010%	0.0433%	0.3660%
fmm	6.92%	27.38%	35.24%	0.3968%	1.8503%	0.0136%	0.0103%
ocean	4.71%	3.78%	5.71%	0.1126%	0.0923%	0.0029%	0.0010%
water-spatial	26.25%	29.33%	29.66%	2.7904%	2.7860%	0.0056%	0.0059%
kmeans	2.24%	0.15%	0.06%	0.0003%	0.0002%	0.0003%	0.0002%
blackscholes	0.46%	0.03%	0.03%	0.0312%	0.0253%	0.0312%	0.0253%
bodytrack	32.67%	83.26%	83.26%	19.192%	19.192%	6.4751%	6.4751%
canneal	5.98%	34.61%	37.40%	6.2946%	10.095%	0.0353%	0.0279%
fluidanimate	25.66%	37.93%	24.19%	2.2900%	1.4139%	0.0010%	0.0010%
raytrace	2.41%	7.56%	6.60%	0.2730%	0.2217%	0.0076%	0.0060%
swaptions	0.15%	0.01%	0.002%	0.0092%	0.0028%	0.0092%	0.0027%
streamcluster	7.03%	87.09%	98.39%	30.274%	78.775%	0.0685%	0.0751%
Average	9.33%	27.99%	39.15%	7.7378%	13.294%	3.5458%	3.6228%

that most of the multi-sharer entries have 2 or 3 sharers. As explained above, when cache size increases, many private entries become shared, but most of them exhibit only 2- or 3-way sharing. The coverage for entries with 4 or more sharers remain small, even at the max PRD. To illustrate, the fourth and fifth columns in Table 6.4 report the percentage of entries for "≥4 sharers." As Table 6.4 shows, at 1MB, 27.99% of the entries are multi-sharer entries, while only 7.74% of the entries have more then 4 sharers. Therefore, 72.36% of the multi-shared entries have 2 or 3 sharers. Even at the maximum PRD, only 13.29% of the entries have more then 4 sharers, showing that 66.43% of the multi-shared entries have 2 or 3 sharers. As discussed above, at the maximum PRD, all of the sharing in the benchmarks are exposed. Among all 15 benchmarks, only three of them, lu, barnes and streamscluster have significant sharing.

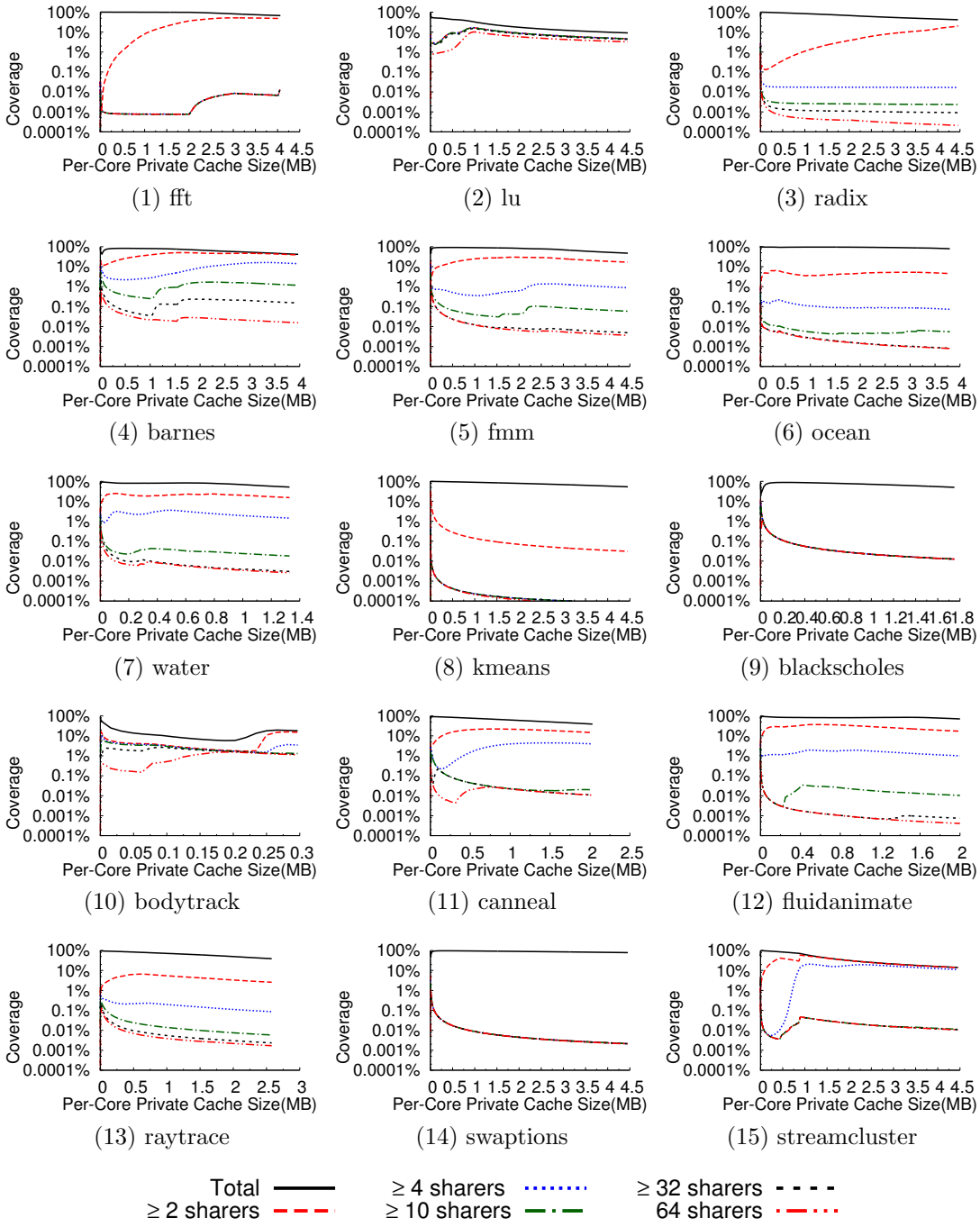


Figure 6.4: Entries with wide sharing for 64-core CPUs

To illustrate further, Figure 6.4 plots the coverage for widely shared entries, using a log scale on the Y-axis. For example, as Figure 6.4 shows, directory entries with ≥32 entries account for very a small fraction of coverage across all cache sizes. And the increase with cache size scaling is insignificant. Table 6.4 also reports the percentage of entries for "≥32 sharers." In 13 out of 15 benchmarks, directory entries with ≥32 sharers account for less than 0.07% of the total entries at 1MB cache, and less then 0.4% at the maximum PRD. Therefore, *the reduction in coverage with cache size scaling mostly comes from the increase of narrowly shared entries, instead of widely shared entries.*

6.3.2 Core Count Scaling

Similar to study 1 in Section 6.2, core count scaling also affects the directory content. As Section 6.2 shows, core count scaling increases sharing in private caches, so it will decrease the directory coverage too. Figure 6.5 illustrates the impact of core count scaling on coverage by plotting the total coverage for 16, 64 and 256 cores. Also the x axis in this figure is total cache size instead of per-core cache size as in Figure 6.3 to permit comparisons across core count.

As Figure 6.5 shows, core count scaling's impact on coverage is not as big as cache size scaling, which also matches the results in study 1. To illustrate, Table 6.5 reports the change in directory coverage from 64 cores to 16 cores and to 256 cores at two different cache sizes, 256KB and 1MB. As Table 6.5 shows, the changes are not very significant. At 256KB, the change in coverage on average is only 4.20%

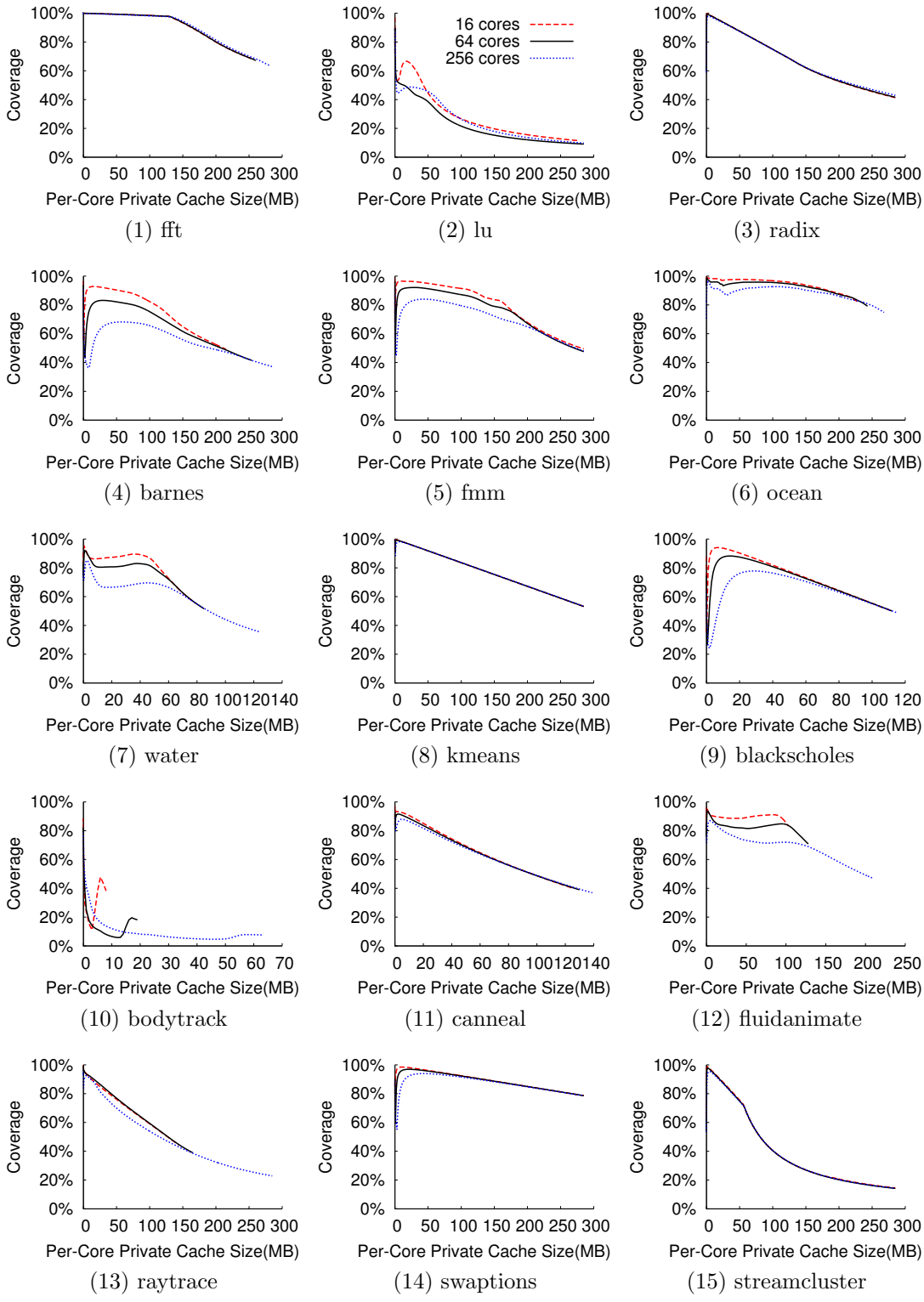


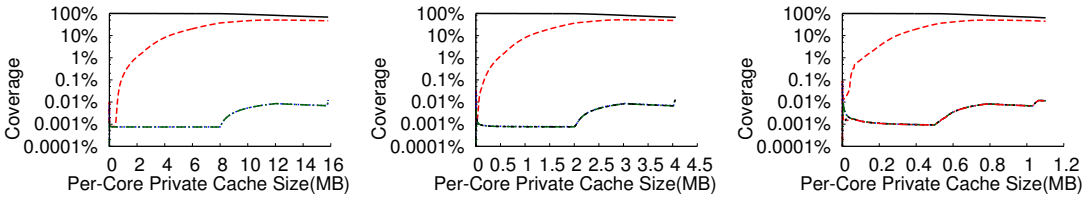
Figure 6.5: Coverage *vs.* private data cache size for 16-, 64-, and 256-core CPUs

Table 6.5: Coverage drop in coverage due to core count scaling.

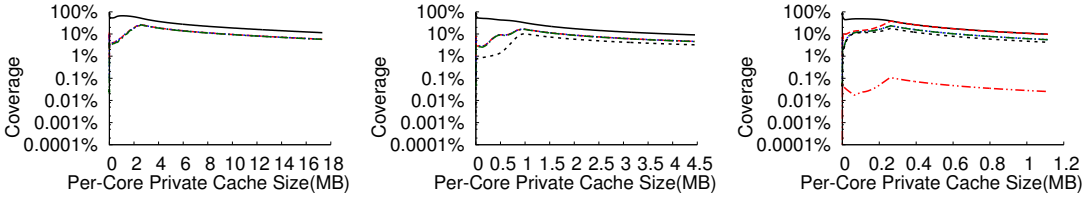
Benchmark	16 Cores		256 Cores	
	256KB	1MB	256KB	1MB
fft	-0.04%	-0.04%	0.18%	0.20%
lu	-17.14%	-4.86%	1.13%	-8.43%
radix	-0.08%	-0.03%	0.60%	0.22%
barnes	-11.26%	-8.05%	27.23%	12.95%
fmm	-4.71%	-3.83%	11.84%	7.07%
ocean	-2.25%	-1.80%	4.64%	4.08%
water-spatial	-6.35%	0.01%	14.11%	3.93%
kmeans	-0.03%	0.02%	0.18%	-0.05%
blackscholes	-3.63%	-0.44%	14.78%	1.81%
bodytrack	-20.24%	-20.14%	8.88%	10.50%
canneal	-1.77%	-0.47%	1.94%	0.38%
fluidanimate	-5.28%	-8.07%	2.28%	10.35%
raytrace	1.49%	0.57%	1.80%	6.90%
swaptions	-1.55%	-0.32%	6.23%	1.33%
streamcluster	-0.47%	0.06%	0.60%	-0.31%
Average	-1.58%	-0.47%	2.62%	1.79%

from 16 cores to 256 cores. While at 1MB, the change in coverage on average is only 2.26% from 16 cores to 256 cores. With a 16x change in core count, the change in coverage is within 5%, which is much less than the cache size scaling effect. In a few cases, the change is over 20%, but still smaller compared to the effect of cache size scaling.

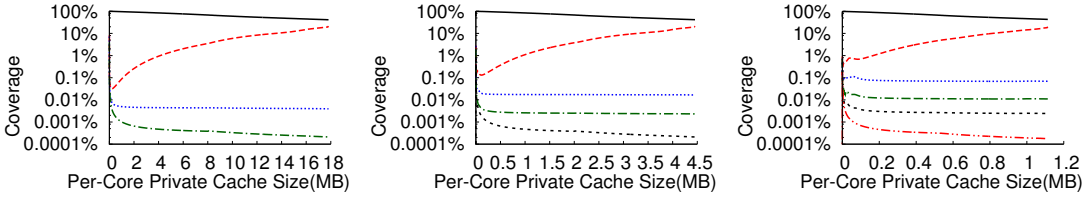
To illustrate how core count scaling affects the coverage, Figure 6.4 shows the coverage breakdown for all benchmarks. Figure 6.4 plots the exact same results as in Figure 6.4, except showing the complete set for 16, 64 and 256 cores. Therefore the 64-core parts of Figure 6.4 is the same as Figure 6.4. Figure 6.4 shows, core count affects the widely sharing entries. In many benchmarks, the directory entries that have 16 sharers when the application runs with 16 threads scales to 64 sharers when the application runs with 64 threads and to 256 sharers when the application runs



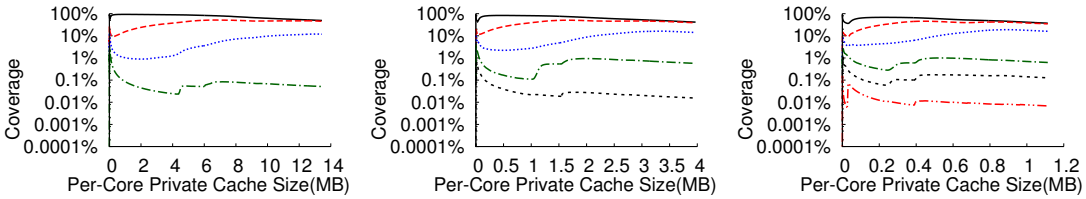
(1) fft,16-cores (2) fft,64-cores (3) fft,256-cores



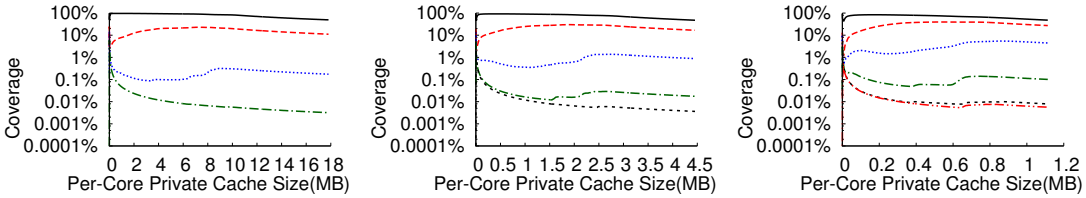
(4) lu,16-cores (5) lu,64-cores (6) lu,256-cores



(7) radix,16-cores (8) radix,64-cores (9) radix,256-cores

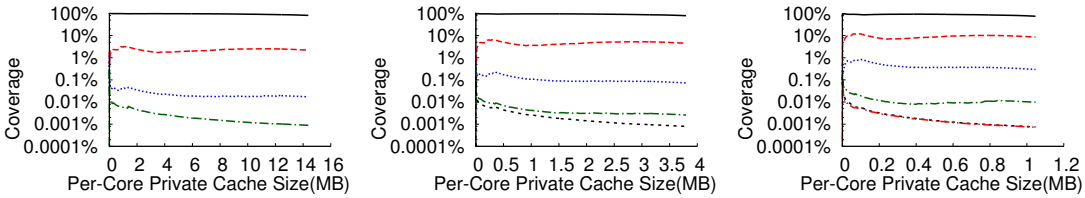


(10) barnes,16-cores (11) barnes,64-cores (12) barnes,256-cores



(13) fmm,16-cores (14) fmm,64-cores (15) fmm,256-cores

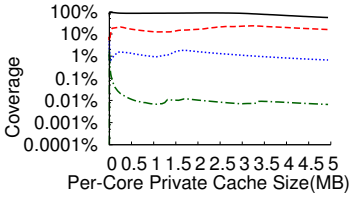
Figure 6.6: Entries with wide sharing for 16-, 64-, 256-core CPUs



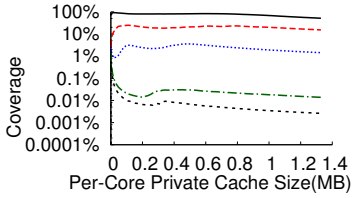
(16) ocean,16-cores

(17) ocean,64-cores

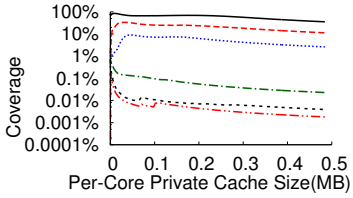
(18) ocean,256-cores



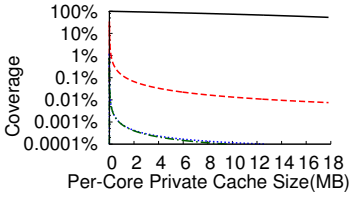
(19) water,16-cores



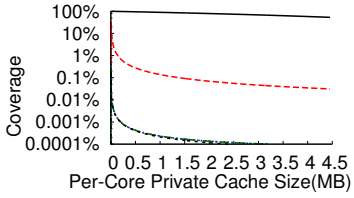
(20) water,64-cores



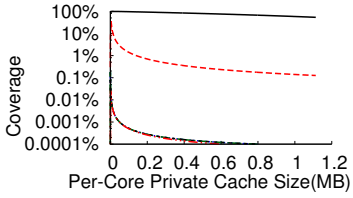
(21) water,256-cores



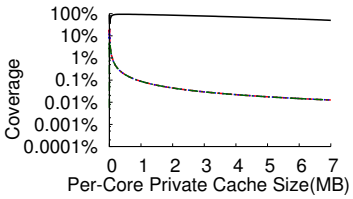
(22) kmeans,16-cores



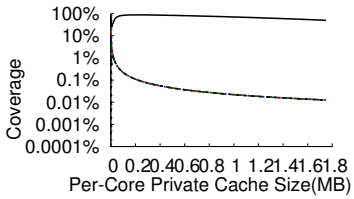
(23) kmeans,64-cores



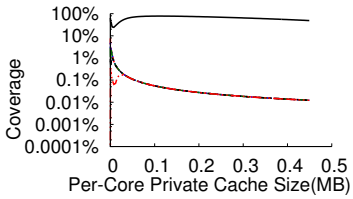
(24) kmeans,256-cores



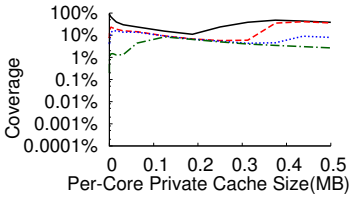
(25) blackscholes,16-cores



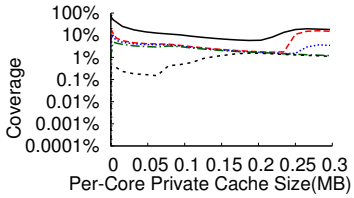
(26) blackscholes,64-cores



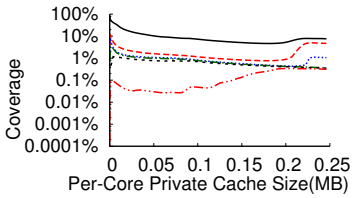
(27) blackscholes,256-cores



(28) bodytrack,16-cores

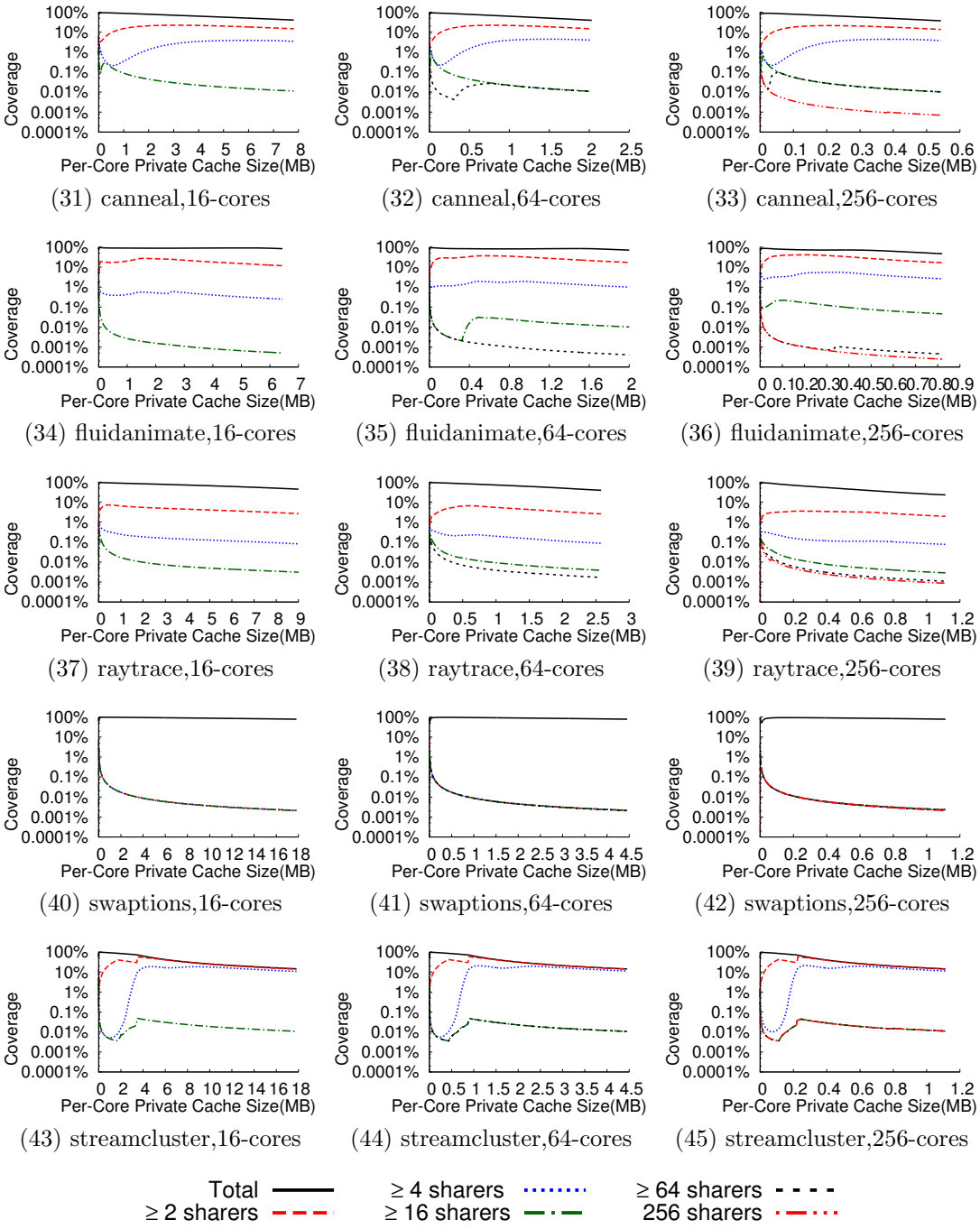


(29) bodytrack,64-cores



(30) bodytrack,256-cores

Figure 6.5: (Continued) Entries with wide sharing for 16-, 64-, 256-core CPUs



(31) canneal,16-cores (32) canneal,64-cores (33) canneal,256-cores

(34) fluidanimate,16-cores (35) fluidanimate,64-cores (36) fluidanimate,256-cores

(37) raytrace,16-cores (38) raytrace,64-cores (39) raytrace,256-cores

(40) swaptions,16-cores (41) swaptions,64-cores (42) swaptions,256-cores

(43) streamcluster,16-cores (44) streamcluster,64-cores (45) streamcluster,256-cores

Figure 6.4: (Continued) Entries with wide sharing for 16-, 64-, 256-core CPUs

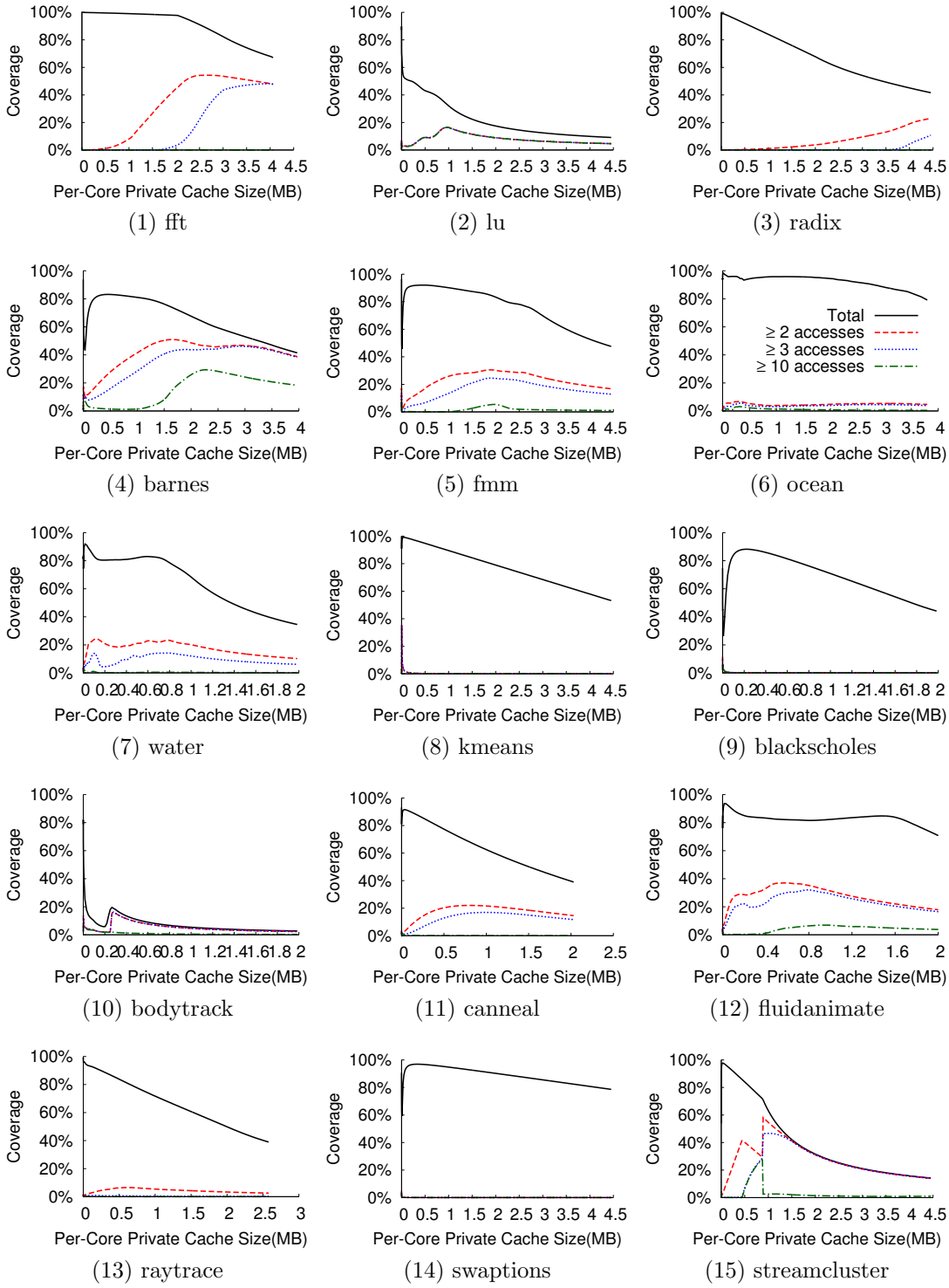
with 256 threads. Therefore, the number of sharers for these entries increases when core count increases, so the coverage generally drops with increasing core count. However, the fraction of these entries is very small. As discussed above, most multi-shared entries only have 2 or 3 sharers. And most of these entries are not affected by the core count. Therefore, the overall impact of core count scaling on coverage is small.

6.4 Study 3: Directory Access Distribution

6.4.1 Cache Size Scaling

Lastly, this thesis studies how accesses are distributed across different individual directory entries. To illustrate, Figure 6.5 breaks down the coverage of directory entries by number of accesses they receive during their life time in the directory cache, as reported by the "dir entry access ctrs" from the profiler. The graphs are formatted in the same fashion as Figure 6.3. The difference is that the graphs in Figure 6.3 break down coverage of directory entries in terms of number of sharers rather than number of accesses. In particular, Figure 6.3 plots the coverage for all entries (labeled "Total"), with ≥ 2, ≥ 3, and ≥ 10 directory accesses for a 64-core CPU.

Comparing the graphs in Figure 6.5 to their corresponding graphs in Figure 6.3 shows many similarities. For example, the coverage for single-access entries (the gap between "Total" and "≥ 2 accesses") in Figure 6.5 is the same as the single-sharer entries (the gap between "Total" and "≥ 2 sharer") in Figure 6.3. This is



(1) fft (2) lu (3) radix

(4) barnes (5) fmm (6) ocean

(7) water (8) kmeans (9) blackscholes

(10) bodytrack (11) canneal (12) fluidanimate

(13) raytrace (14) swaptions (15) streamcluster

Figure 6.5: Distribution access during their lifetimes

because single-sharer entries are private entries, and private entries only get one access during their lifetimes–the access that fills the entry into the directory cache. So, the coverage for multi-access entries (≥ 2 accesses) in Figure 6.5 is the same as for multi-sharer entries (≥ 2 sharers) in Figure 6.3.

Moreover, for directory entries with higher sharing degree, the access counts are also related to their sharing degree. To be specific, the access counts of an entry is equal to or greater than its sharing degree. Therefore, as was discussed in Section 6.3, that sharing increases non-uniformly with cache size scaling, so too does the access distribution increase non-uniformly. As illustrated in Figure 6.5, *A small part of the directory receives a disproportionately large fraction of the directory accesses.*

Table 6.6 quantifies this phenomenon. The second to fourth columns of Table 6.6 report the portion of accesses to the entries with ≥ 3 accesses during their lifetimes at 256KB, 1MB, and ∞ private caches. And the fifth to seventh columns report the percentage of directory entries with ≥ 3 accesses during their lifetimes. At 256KB, Table 6.6 shows the entries with ≥ 3 accesses during their lifetimes account for only 5.4% of all directory entries, but receive 23.2% of all directory accesses on average. At 1MB, such entries account for 23.0% of directory entries but receive 41.6% of the total accesses. And at ∞ cache, they account for 35.2% of directory entries but receive 57.0% of the total accesses.

In addition, the study looks at the portion of T2 accesses to these entries. As discussed in Section 4.1, T1 accesses are cache transactions that miss all the way to the sharing point, while T2 accesses are directory reuse transactions. Therefore, T2

Table 6.6: Percent accesses destined to $\geq$ 3-access entries, percent entries with $\geq$ 3 accesses, and percent T2 accesses destined to $\geq$ 3-access entries.

Benchmark	% Accesses to $\geq$3 Entries			% Entries with $\geq$3 Accesses			% T2 Accesses to $\geq$3 Entries	
	256KB	1MB	∞	256KB	1MB	∞	256KB	1M
fft	0.4%	0.5%	84.8%	0.0%	0.0%	71.1%	70.8%	10.3%
lu	54.5%	98.1%	98.4%	8.2%	51.0%	50.9%	100.0%	100.0%
radix	1.8%	1.8%	33.1%	0.1%	0.1%	25.9%	91.3%	55.0%
barnes	29.4%	52.9%	99.2%	12.3%	35.1%	92.5%	88.2%	86.6%
fmm	9.0%	26.8%	62.1%	5.2%	14.9%	27.0%	72.7%	76.4%
ocean	13.4%	27.9%	44.4%	5.2%	3.3%	4.9%	95.5%	97.5%
water-spatial	11.4%	43.5%	45.5%	6.3%	17.6%	17.6%	61.6%	83.8%
kmeans	55.9%	55.9%	55.9%	0.5%	0.1%	0.1%	100.0%	100.0%
blackscholes	0.8%	0.8%	0.8%	0.1%	0.0%	0.0%	100.0%	100.0%
bodytrack	80.8%	99.3%	99.3%	11.4%	85.2%	85.2%	99.7%	100.0%
canneal	14.2%	42.9%	51.2%	7.5%	27.1%	30.1%	65.5%	88.8%
fluidanimate	38.6%	63.9%	69.4%	23.4%	35.5%	23.3%	89.1%	98.6%
raytrace	4.1%	4.4%	4.4%	0.8%	0.6%	0.5%	59.7%	43.4%
swaptions	6.3%	6.3%	6.3%	0.0%	0.0%	0.0%	100.0%	100.0%
streamcluster	26.9%	98.9%	99.9%	0.2%	73.9%	98.4%	70.0%	99.9%
Average	23.2%	41.6%	57.0%	5.4%	23.0%	35.2%	84.3%	82.7%

accesses are always on-chip transactions while T1 accesses can be off-chip transactions if the sharing point is at the chip boundary. Hence T2 directory accesses are more latency sensitive. The eighth to ninth columns of Table 6.6 report the portion of reuse transaction to entries with $\geq$ 3 accesses during their lifetimes at 256KB and 1MB private caches. Table 6.6 shows that at 256KB, 5.4% of the directory entries received 84.3% of the T2 accesses while at 1MB 23.0% of the directory entries received 82.7% of the T2 accesses. Therefore, *a small fraction of the directory cache not only receives a large fraction of the total directory accesses, but also receives the majority of latency-sensitive directory accesses.*

When a directory cache is implemented, it is usually sized using a fixed coverage over data cache size. Therefore, this study also looks at the accesses' locality

Table 6.7: Percentage of accesses towards directory of 18.75% coverage

Benchmark	% T2 Accesses		% Total Accesses	
	256KB	1MB	256KB	1MB
fft	100.0%	100.0%	19.3%	19.3%
lu	100.0%	100.0%	69.3%	98.4%
radix	100.0%	100.0%	21.1%	24.8%
barnes	100.0%	73.5%	37.0%	42.5%
fmm	100.0%	86.5%	22.3%	32.2%
ocean	100.0%	100.0%	26.2%	40.6%
water-spatial	98.7%	97.4%	21.6%	54.5%
kmeans	100.0%	100.0%	64.2%	65.1%
blackscholes	100.0%	100.0%	21.9%	27.1%
bodytrack	100.0%	100.0%	100.0%	100.0%
canneal	100.0%	93.2%	30.0%	46.0%
fluidanimate	86.4%	87.0%	37.2%	54.2%
raytrace	100.0%	100.0%	24.4%	31.9%
swaptions	100.0%	100.0%	24.4%	24.8%
streamcluster	93.9%	97.4%	45.1%	95.7%
Average	98.6%	95.7%	37.6%	50.5%

with fixed coverage. Figure 6.6 shows profiling results. The profiler ranks the directory entries with the number of accesses they receive during their lifetimes, then finds out the first N entries that occupy 18.75% coverage and computes the number of total and T2 accesses towards those entries. Figure 6.6 shows that with a directory cache sized of 18.75% coverage receives a large fraction of the total directory accesses, and the majority of T2 directory accesses. Table 6.7 reports that at 256KB, directory cache with 18.75% coverage receives 98.6% of the T2 directory accesses and 37.6% of the total directory accesses; and at 1M, it receives 95.7% T2 directory accesses and 50.5% of the total directory accesses. In most cases, this high percentage is due to the locality in the directory cache. In some other cases, such as lu, bodytrack and streamclutser, this high percentage is because the total coverage is lower than 18.75%.

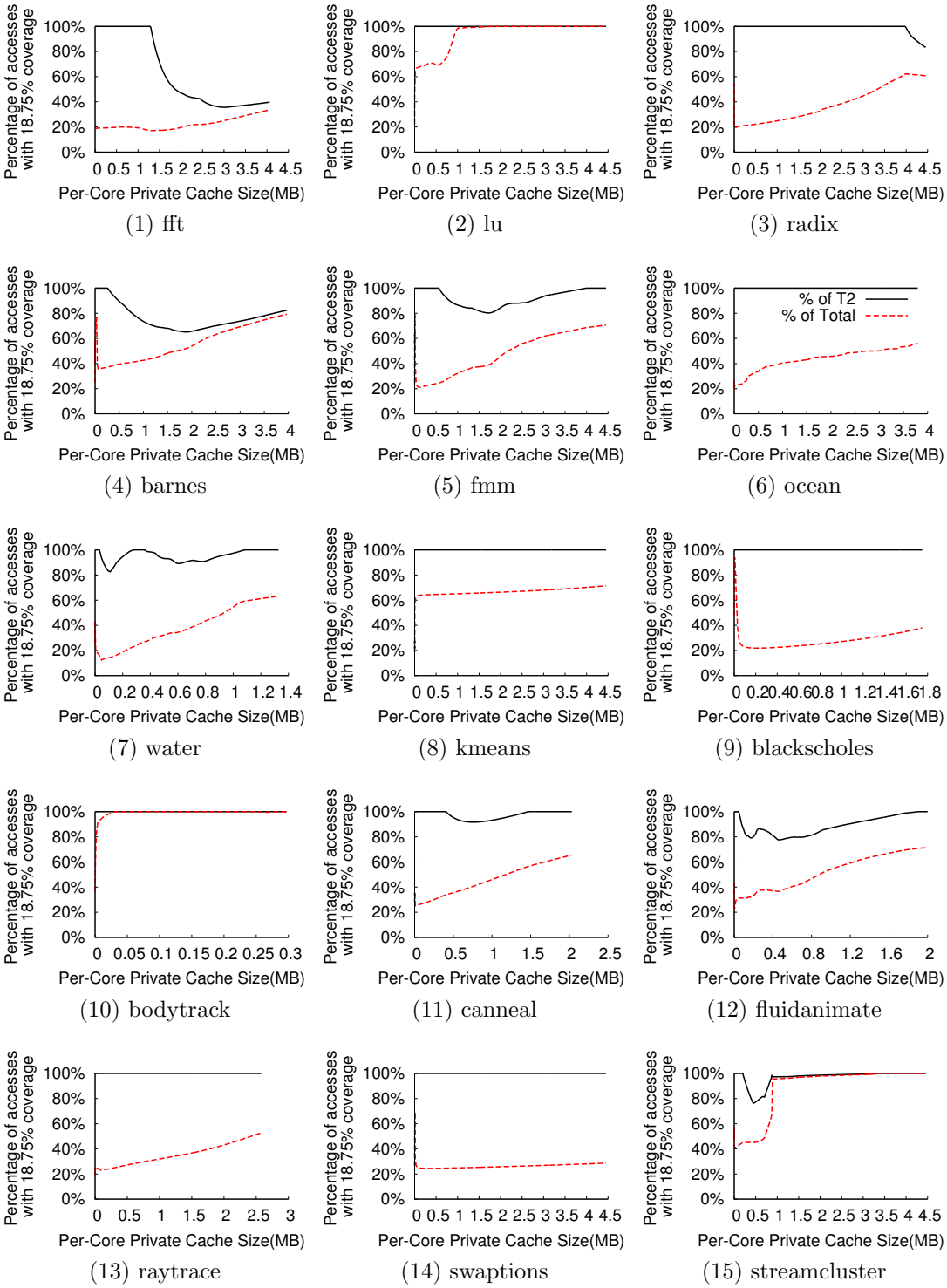


Figure 6.6: Percentage of accesses towards directory of 18.75% coverage

Table 6.8: Percentage Difference in accesses destined to ≥ 3-access entries, entries with ≥ 3 accesses, and T2 accesses destined to ≥ 3-access entries for 16-cores and 256-cores from 64-cores, at 64MB.

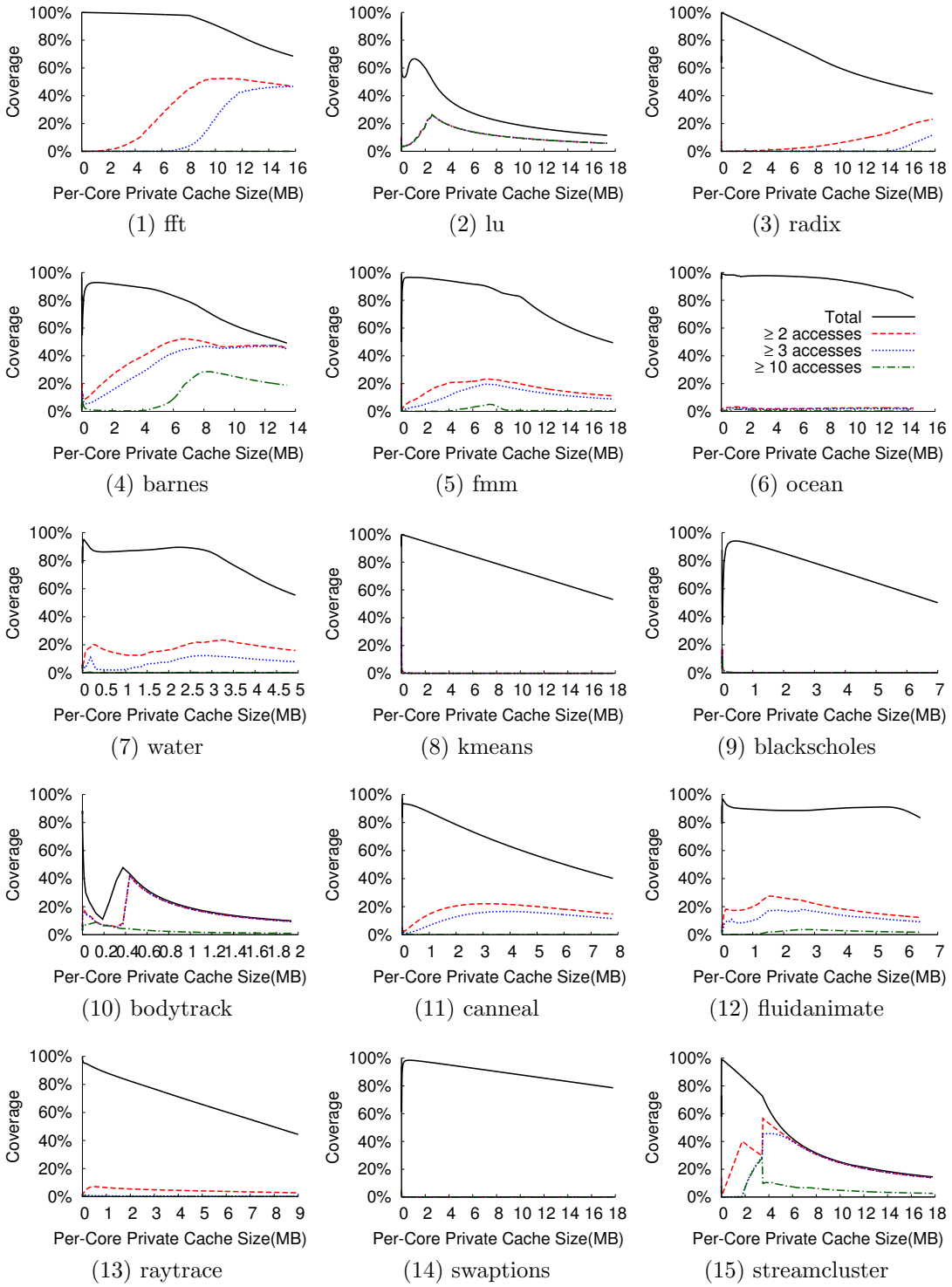
Benchmark	% Accesses to ≥3 Entries		% Entries with ≥3 Accesses		% T2 Accesses to ≥3 Entries	
	16-cores	256-cores	16-cores	256-cores	16-cores	256-cores
fft	-0.3%	1.3%	0.0%	0.0%	-7.6%	24.6%
lu	-3.2%	-8.7%	0.1%	17.7%	0.0%	-0.6%
radix	-1.6%	2.9%	-0.1%	0.3%	-43.7%	16.8%
barnes	-5.4%	2.7%	-3.3%	1.5%	-1.6%	0.9%
fmm	-4.4%	7.1%	-2.9%	4.9%	-0.4%	3.3%
ocean	-14.3%	19.5%	-1.9%	3.5%	-1.3%	0.6%
water-spatial	-5.7%	8.3%	-3.3%	8.0%	-7.2%	8.3%
kmeans	-1.2%	-37.7%	-0.1%	0.5%	0.0%	0.0%
blackscholes	-0.6%	2.5%	0.0%	0.0%	0.0%	0.0%
bodytrack	0.5%	-0.5%	10.5%	-22.5%	0.0%	0.0%
canneal	-2.0%	0.6%	-0.8%	-0.9%	-1.3%	0.4%
fluidanimate	-23.6%	8.5%	-20.5%	17.0%	-5.6%	-0.3%
raytrace	-0.2%	0.3%	-0.1%	0.2%	-1.1%	19.9%
swaptions	-2.2%	5.6%	0.0%	0.0%	0.0%	0.0%
streamcluster	0.4%	-0.1%	-1.5%	-0.2%	0.0%	0.0%
Average	-4.3%	0.8%	-1.6%	2.0%	-4.6%	4.9%

6.4.2 Core Count Scaling

Similar to study 2 in Section 6.3, core count scaling also affects the directory access distribution. As explained above, the directory access distribution has a close relationship with the directory sharing degree; therefore, core count scaling has a similar effect on directory access distribution.

To illustrate, Figures 6.7 and 6.8 plot the access distribution profile for 16 and 256 cores. The graphs are in the same format as Figure 6.5. Comparing Figure 6.7, 6.8 and 6.5 show these three graphs are very similar. Similar to Study 1 and Study 2, core count scaling has a small effect on access distribution too.

To further quantify this phenomenon, Table 6.8 shows the percentage differ-



(1) fft (2) lu (3) radix

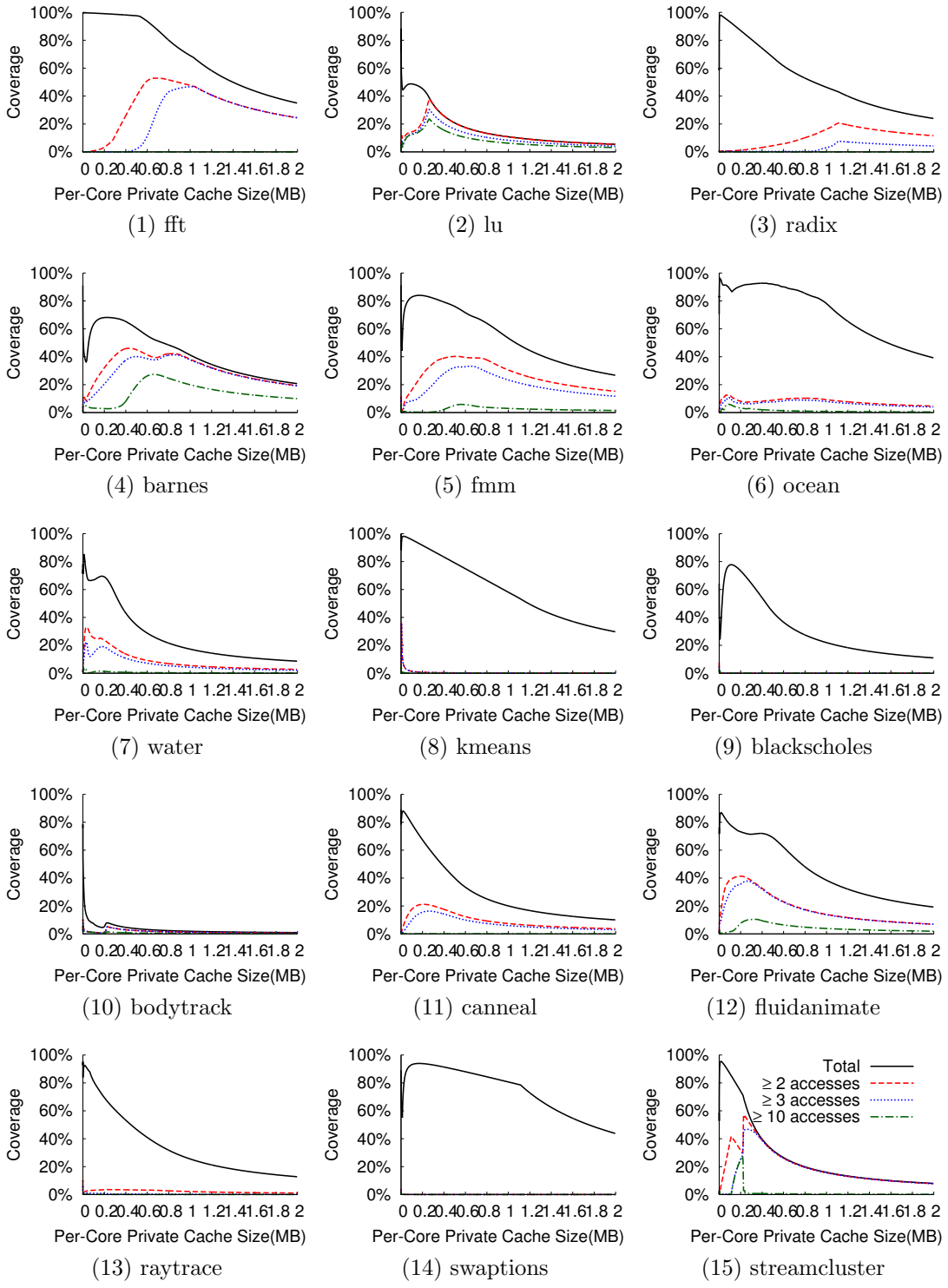
(4) barnes (5) fmm (6) ocean

(7) water (8) kmeans (9) blackscholes

(10) bodytrack (11) canneal (12) fluidanimate

(13) raytrace (14) swaptions (15) streamcluster

Figure 6.7: Distribution access during their lifetimes for 16-cores



(1) fft (2) lu (3) radix

(4) barnes (5) fmm (6) ocean

(7) water (8) kmeans (9) blackscholes

(10) bodytrack (11) canneal (12) fluidanimate

(13) raytrace (14) swaptions (15) streamcluster

Figure 6.8: Distribution access during their lifetimes for 256-cores

ence in accesses destined to $\geq$ 3-access, in entries with $\geq$ 3 accesses, and in T2 accesses destined to $\geq$ 3-access entries at 64MB total private cache, when comparing 16 cores and 256 cores against 64 cores. As Table 6.8 shows, the percentage of accesses, entries and T2 accesses generally increase with core count scaling, but the change is very small. From 16-cores to 256 cores, the increase in percentage of accesses destined to $\geq$ 3-access is only 8.9%, the increase in percentage of entries with $\geq$ 3-access is only 3.6%, and the increase in percentage of T2 accesses destined to $\geq$ 3-access is only 5.7%, which is quite small considering there is a 16x increase in core count.

Chapter 7: Cache Simulations and Validations

The profiler uses LRU stacks to model a cache, thus assuming full associativity. Therefore, though the profiler captures the capacity misses and sharing effects, other cache effects, such as conflicts, are not captured. These conflict misses can affect the directory behavior by changing the directory access stream. This chapter uses a cache simulator to quantify the error in the profiling results, showing they are accurate to provide directory behavior insights.

7.1 Experimental Setup

A cache simulator is implemented to model the cache hierarchy in Figure 4.1, using the same PIN tool from Section 5. In the cache simulator, the LRU stacks

Table 7.1: Data and directory cache parameters for simulation validation.

Private Data Cache Sizes (Associativities)	
Private L1:	16KB (4-way)
Private L2:	64KB (8-way)
Private L3:	256KB, 512KB, 1MB, or 2MB (8-way) (64 cores) 4MB (8-way) (16 cores) 256KB(8-way) (256 cores)
Directory Cache Coverage (Associativities)	
Cuckoo:	200% (4-way)

are replaced with data cache models and a directory model. The data cache model implements three levels of private cache, L1, L2 and L3 per core with 64-byte cache blocks. The private caches are inclusive and a MESI protocol is used to maintain cache coherence. This chapter performs validation of the profile predictions from Chapter 6 at four different L3 cache sizes at 64 cores for cache scaling, and also at three different core counts at a total L3 capacity of 64MB. Table 7.1 specifies the cache parameters used in the cache simulations.

In the directory cache model, a Cuckoo Directory [19] is implemented. A Cuckoo directory uses multiple hash functions and iterative re-insertion to increase the effective associativity of the directory cache. Re-insertion is limited to 32 attempts in this study. In the validation experiments, the Cuckoo directory is over-provisioned to 2x the number of directory entries compared to private data cache blocks to reduce the conflicts in the directory. Also, full-map directory entries are used in the experiments, which mirrors the profiler because the profiler tracks all sharers precisely. Table 7.1 specifies the directory cache parameters used in the cache simulations.

7.2 Study 1: Directory Access Frequency

This section validates the directory access analyses in Section 6.2. To illustrate the errors between the profiling results and the simulation results, Figure 7.1 plots the simulation results along with the profiling results for the directory APKI at the 4 different cache sizes from Table 7.1, labeled as "Sim with notifications," "Sim total

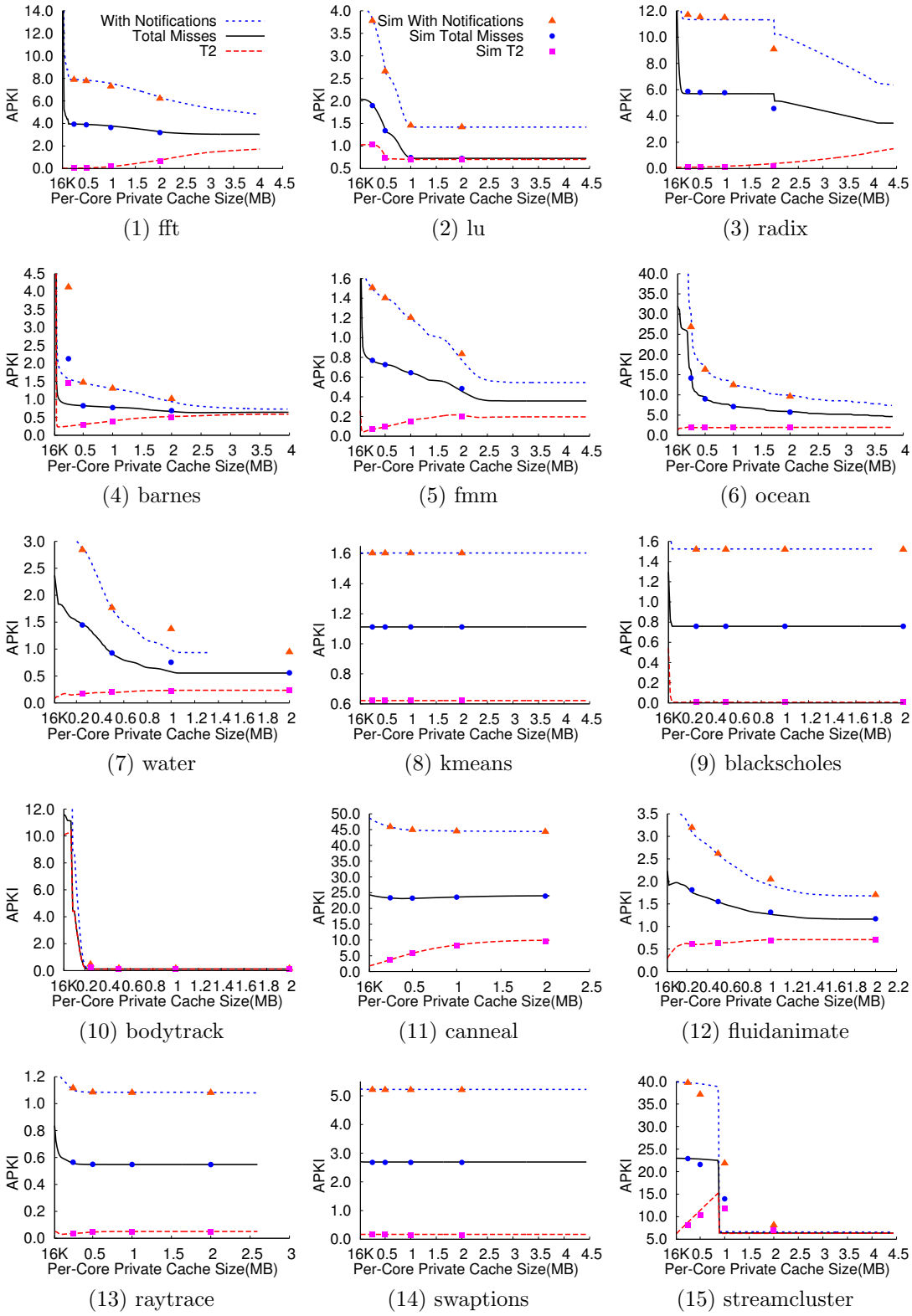
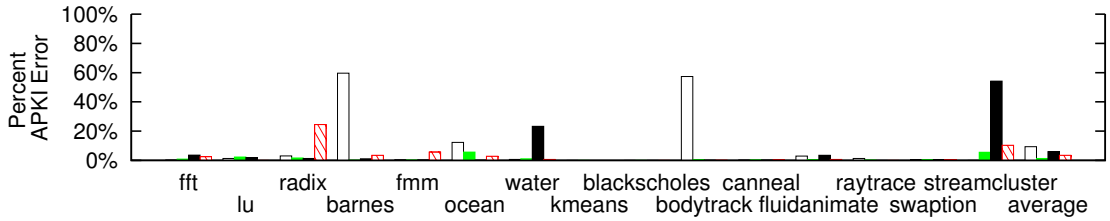
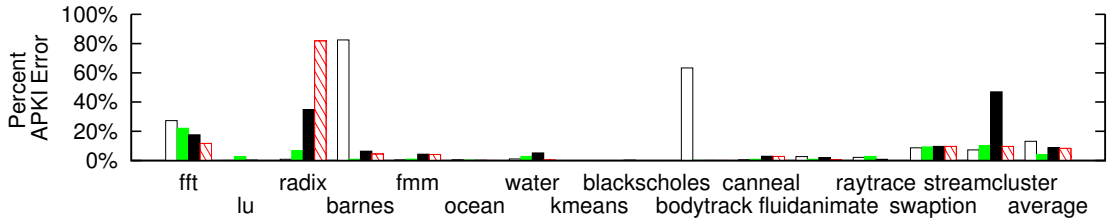


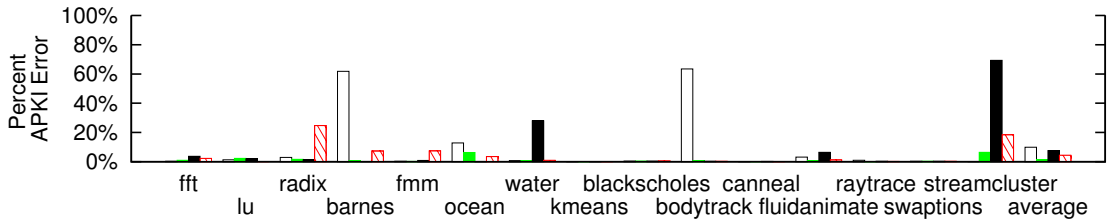
Figure 7.1: Breakdown of directory APKI *vs.* private data cache size for 64-core CPUs



(1) Cache-miss induced directory accesses (T1+T2)



(2) T2 directory accesses



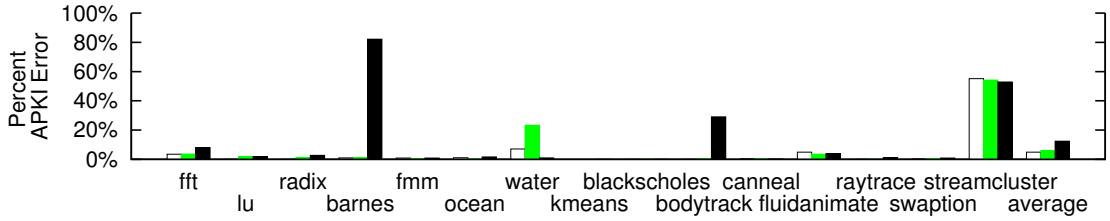
(3) Total directory accesses with notifications

Figure 7.2: Percent APKI error for directory accesses.

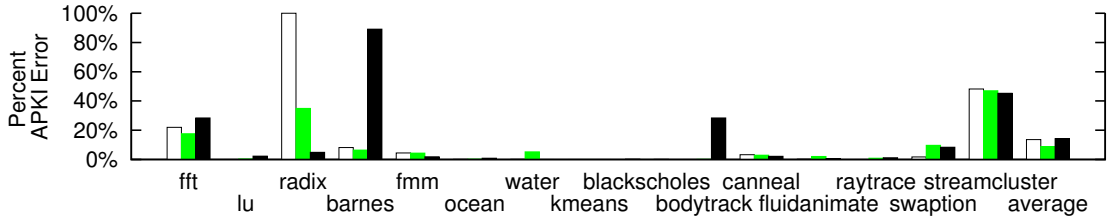
misses," "Sim T2." All results are for 64 cores. Figure 7.1 shows visually the errors in total cache-miss induced directory accesses, T2 accesses and directory accesses with notifications that the profiler predicts.

To show the results quantitatively, Figure 7.2 plots the percentage error for directory accesses between the profiling results and simulation. Figure 7.2(1) plots the percentage error for cache miss-induced directory accesses, Figure 7.2(2) plots the percentage error for T2 directory accesses, and Figure 7.2(3) plots the percentage error for directory accesses with notifications. As Figure 7.2 shows, the simulation and profiling results are very close in most cases. The profiling results are within 7% of simulation, for 88% of the data points in total cache miss-induced APKI, for 73% of the data points in T2 APKI, and for 85% of the data points in directory accesses with notifications, respectively. Across all benchmarks in the 64-core validations, the error is 5.0% for total cache miss-induced APKI, 8.6% for T2 APKI and 5.7% for directory accesses with notifications, respectively.

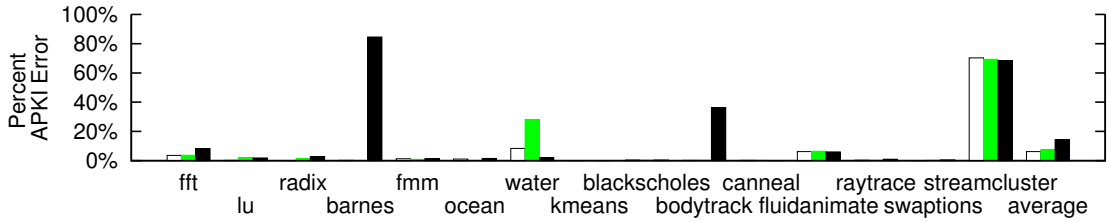
There are validation points in Figure 7.2 that have high error, but most of these are benign. Figure 7.1 gives a visualization of the source of these errors. In one case, the high error is due to the very small APKI. As discussed in Section 6.2, directory access frequency drops with cache size scaling, making APKI very small for some benchmarks at certain cache sizes. In this case, tiny absolute errors can result in large percentage error. This happens in bodytrack as shown in Figure 7.1(10) and radix's T2 access as shown in Figure 7.1(3). Another case is the sudden drop in the access frequency. Because of conflict misses, the profiler may mis-judge the capacity at which such drops happen slightly, which can result in large error. This happens



(1) Cache-miss induced directory accesses (T1+T2)



(2) T2 directory accesses



(3) Total directory accesses (T1+T2+Notifications)

Figure 7.3: Percent APKI error for directory accesses for 16, 64, 256 cores.

in barnes as shown in Figure 7.1(4) and streamcluster as shown in Figure 7.1(15). But these errors only occurs locally at capacities near the drop.

This section also validates the profiling result across multiple core counts. Figure 7.3 reports the error for 16, 64, 256 cores for total cache-miss induced directory accesses, T2 accesses and directory accesses with notifications. Similar to Figure 7.2, the errors are low. The profiling results are within 7% of simulation for 82% of the data points in total cache miss-induced APKI, for 71% of the data points in T2 APKI, and for 82% of the data points in directory accesses with notifications, respectively. Across all benchmarks in the core count validations, the error is 8.0% for total cache miss-induced APKI, 12.2% for T2 APKI and 9.4% for directory accesses

with notifications, respectively. Also, similar to the cache size scaling results for 64 cores, most of the cases with elevated errors in the core count scaling results are benign.

Overall, the simulation results show the profiler can predict directory cache accesses with good accuracy.

7.3 Study 2: Directory Coverage

In addition to directory accesses, this section also quantifies the error in the directory content analyses in Section 6.3. The cache simulator measures the average number of live directory entries and the average number of live shared directory entries (entries with ≥ 2 sharers) in the simulated directory cache using a cuckoo directory with 200% coverage, as explained in Section 7.1, because this virtually ensures that there will be no directory induced evictions due to conflicts in the directory. Figure 7.4 plots the simulation results along with the profiling results for the 4 different cache sizes in Table 7.1 at 64 cores to visually illustrate the agreement between the profile results and simulation. Figure 7.4 plots the simulation results for all directory entries' coverage, labeled as "Sim total," and the results for shared directory entries' coverage, *i.e.*, entries with equal or more than two sharers, labeled as "Sim ≥ 2 sharers."

To show the results quantitatively, Figure 7.5 plots the percentage error between the simulation and profiling coverage results for 64 cores. Figure 7.5(1) plots the percentage error for all directory entries. Figure 7.5(2) plots the percentage

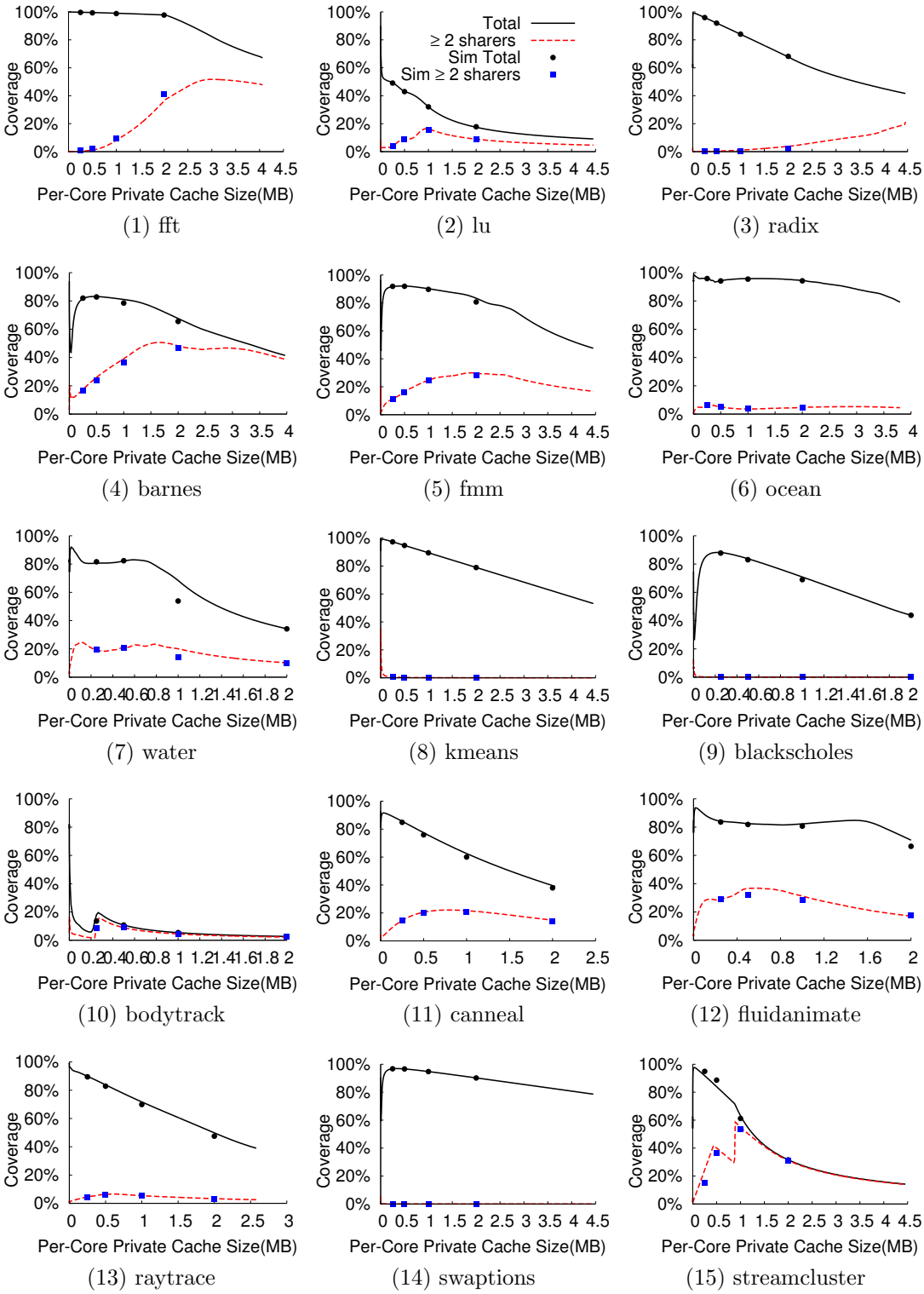
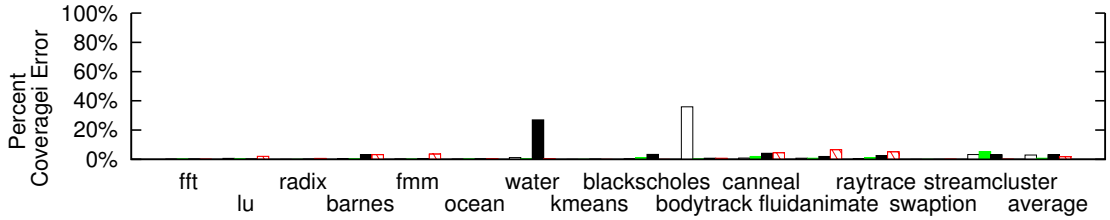
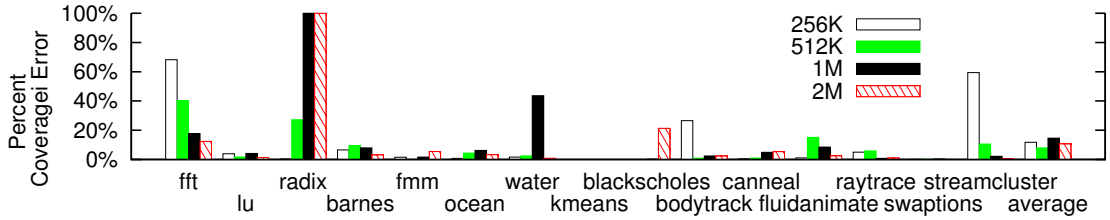


Figure 7.4: Coverage *vs.* private data cache size for 64-core CPUs



(1) All directory entries



(2) Directory entries with ≥ 2 sharers

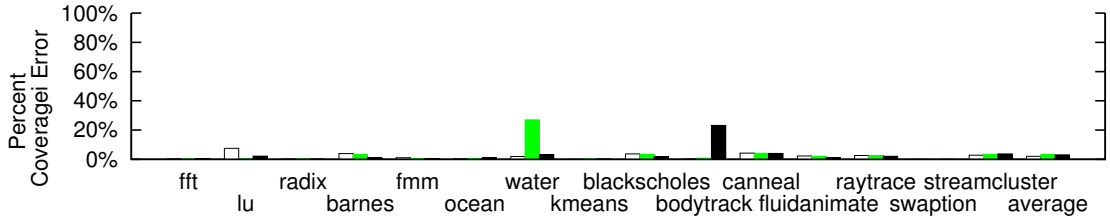
Figure 7.5: Percent coverage error.

error for the shared directory entries.

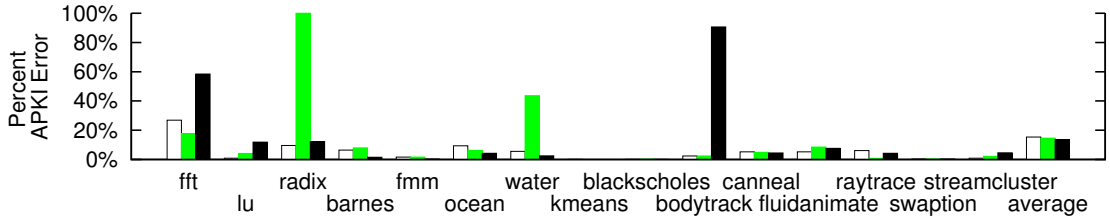
Figure 7.5(1) shows the framework can predict coverage for all directory entries with good accuracy. Across all benchmarks in the 64-core validations, the average coverage error is only 2.2%. Similar to Section 7.2, the main source of error is the conflict misses in the data cache, as the profiler does not capture conflicts in the private data cache. Also similar to Section 7.2, there are some data points with high errors, such as bodytrack. This is also because absolute value is small and thus tiny absolute errors result in large percentage error.

Figure 7.5(2) shows the framework can also predict the coverage for shared entries as well. Across all benchmarks in the 64-core validations, the coverage average error is 11.2%. This is relatively larger than the results for all directory entries. Nevertheless, the profiling results are within 7% of simulation for 73% of the data points.

There are validation points in Figure 7.5(2) that have high error, but most of



(1) All directory entries



(2) Directory entries with ≥ 2 sharers

Figure 7.6: Percent coverage error for 16, 64, 256 cores.

these are benign. Figure 7.4 gives a visualization of the source of these errors. In one case, the high error is due to the very small coverage. As discussed in Section 6.3, only a small fraction of the directory entries is shared in many benchmarks, especially in small data cache capacities. In this case, tiny absolute errors can result in large percentage error. This happens in fft as shown in Figure 7.4(1), radix as shown in Figure 7.4(3) and blackscholes as shown in Figure 7.4(9). Another case is the sudden change in the directory coverage. Because of conflict misses, the profiler may misjudge the capacity at which the sharing changes slightly, which can result in large error. This happens in bodytrack as shown in Figure 7.4(10), and streamcluster as shown in Figure 7.4(15). But these errors only occurs locally at capacities near the changes.

The profiling results across multiple core counts are also validated. Figure 7.6 reports the coverage error for 16, 64, 256 cores with 64MB total L3 caches. Across all benchmarks in the core count validations, the average coverage error for all directory

entries is 2.7% and the average coverage error for shared directory entries is 14.5%. Also, similar to the cache size scaling results for 64 cores, most of the cases with elevated errors in the core count scaling results are benign.

7.4 Study 3: Directory Access Distribution

In addition to directory content analyses, this section also quantifies the error in the access distribution analyses in Section 6.4. The cache simulator measures the average number of live directory entries with $\geq N$ accesses. Similar to Section 7.3, a cuckoo directory with 200% coverage is used. Figure 7.7 plots the simulation results along with the profiling results for 4 different cache sizes in Table 7.1 at 64 cores to visually illustrate the agreement between the profile results and simulation. As explained in Section 6.4, the coverage for the total directory entries and multi-access entries ($\geq$ 2 accesses) in Figure 7.7 is the same as for the total directory entries and multi-sharer entries ($\geq$ 2 sharers) in Figure 7.4. Therefore, Figure 7.7 plots the simulation results for directory entries with $\geq$ 3 accesses, labeled as "Sim $\geq$ 3 accesses," and the results for for directory entries with $\geq$ 10 accesses, labeled as "Sim $\geq$ 10 accesses."

To show the results quantitatively, Figure 7.8 plots the error of the percentage of directory entries with $\geq$ 3 accesses between the simulation and profiling results for 64 cores.

Figure 7.8 shows the framework can predict directory distribution well. Across all validation points at 64 cores, the average coverage error is 8.7%. Similar to

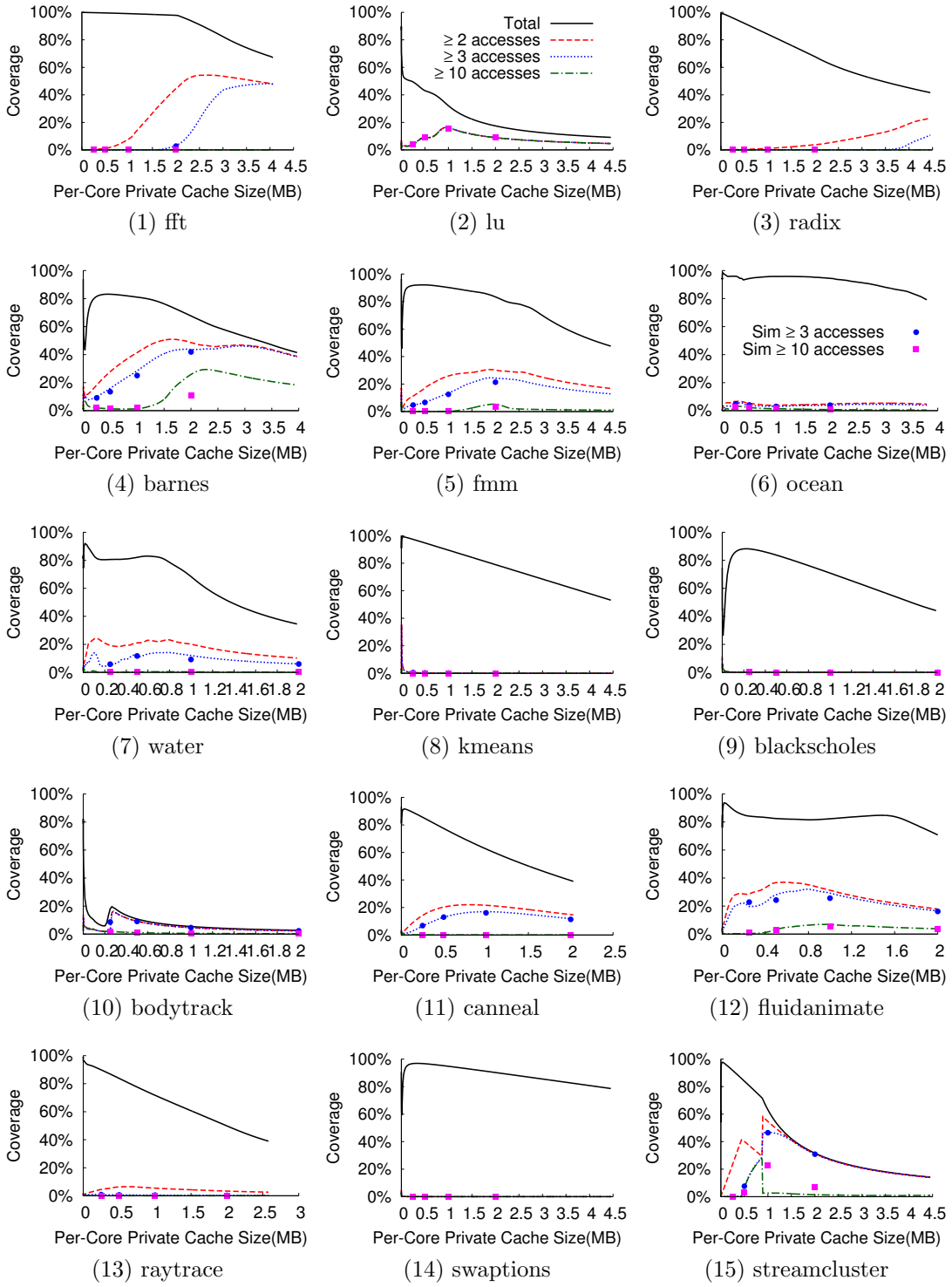


Figure 7.7: Distribution access during their lifetimes

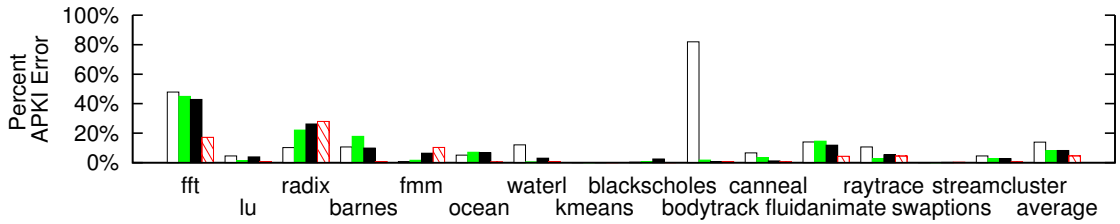


Figure 7.8: Percentage of entries with ≥ 3 accesses

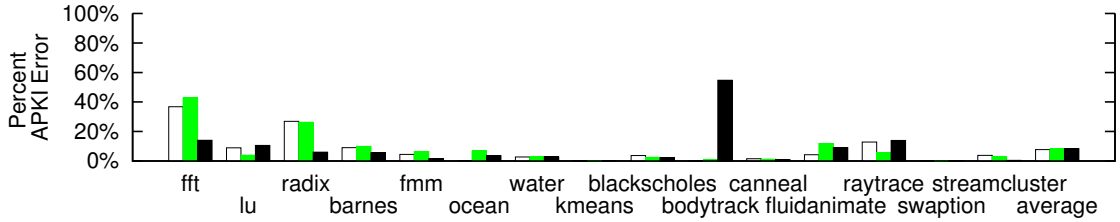


Figure 7.9: Percentage of entries with ≥ 3 accesses for 16, 64, 256 cores

Section 7.4 and 7.3, the main source of error is the conflict misses in the data cache, as the profiler does not capture conflicts in the private data cache. When there is a sudden change in the percentage of directory entries, the profiler may mis-judge the capacity at which the sharing changes slightly, which can result in large error, as shown in bodytrack. Also similar to Section 7.3, many data points have small absolute value, which lead to large percentage error with tiny absolute errors, as shown in fft, radix.

The profiling results across multiple core counts are also validated. Figure 7.9 reports the percentage error for 16, 64, 256 cores with 64MB total L3 caches. Across all benchmarks in core count validations, the average error on percentage of directory entries is 8.1%. Also, similar to the cache size scaling results for 64 cores, most of the cases with elevated errors in the core count scaling results are benign.

Chapter 8: Case Studies and Discussions

This chapter applies the insights that the profiler provides on existing directory techniques, and discusses the implications for such techniques. Experiments are also conducted to verify some of the observations.

8.1 Cuckoo Technique Discussion

Minimizing directory size is an important goal in directory design, but how small a directory can be without penalizing performance is a basic question in directory design. The analyses in Section 4.3 quantify the content of the directory and therefore can help architects make design decisions in sizing directories as CPUs scales.

The first observation in Section 6.3 is that the coverage varies with CPU scaling, especially the drop with cache size scaling, as shown in Figure 6.3. This implies that a smaller fraction of the total on-chip memory is needed for the directory cache as CPUs scale. This affects the techniques that track all directory entries, such as the techniques that focus on reducing the sharers lists, but tracks all entries. To test this implication, A cache simulation study is conducted in this section.

Table 8.1: Data and directory cache parameters for cuckoo experiments.

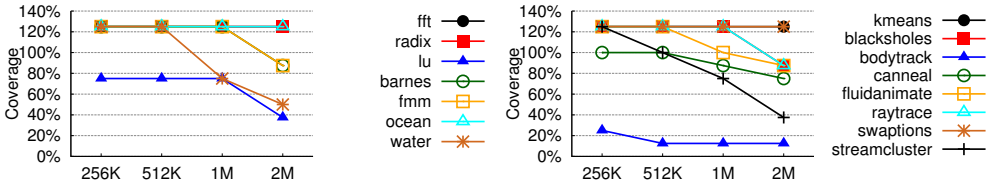
Private Data Cache Sizes (Associativities)	
Private L1:	16KB (4-way)
Private L2:	64KB (8-way)
Private L3:	256KB, 512KB, 1MB, or 2MB (8-way) (64 cores)
Directory Cache Coverage (Associativities)	
Cuckoo:	12.5% (4-way), 25% (4-way), 37.5% (3-way), 50% (4-way), 75% (3-way), 87.5% (7-way), 100% (4-way), 125% (5-way), 200% (4-way),

8.1.1 Experimental Setup

The experimental setup is similar to the one explained in Section 7.1. The simulation is for 64-core CPUs and for 4 different L3 data cache sizes. The upper part of Table 8.1 specifies the cache sizes that are simulated. In this experiment, a Cuckoo directory is simulated. 9 different directory cache sizes are tried to determined the minimum size that achieve good performance. Table 8.1 specifies the cache parameters for the experiments. In particular, The bottom part of Table 8.1 specifies the 9 directory cache sizes (in terms of coverage) that are simulated.

8.1.2 Experiment results

Figure 8.1 plots the minimum Cuckoo directory cache size each benchmark requires in terms of coverage to achieve less then 1% eviction rate. As Figure 8.1 show, Cuckoo directory's coverage drops with data cache scaling, thus implying smaller directories can be used as data caches scale. At 256KB private cache, most benchmarks require 125% coverage. But at 2MB private cache, only five bench-



(1) SPLASH2 (2) PARSEC and MineBench

Figure 8.1: Minimum Cuckoo coverage for 1% eviction rate.

marks remain at 125%. Six benchmarks exhibit coverage between 75–87.5% and four benchmarks drop to 50% or less.

The cuckoo directory coverage results in Figure 8.1 are different from the profiling coverage results in a few ways. First, data cache conflicts are taken into consideration. Second, the conflicts in the directory are also taken into consideration and over-provisioning is used to reduce directory cache conflicts. Third, only a few discrete directory cache sizes are simulated. However, comparing Figures 8.1 and Figure 6.3 still shows the minimum Cuckoo sizes are correlated to the profiling coverage results. In most cases, the Cuckoo coverage is between 30–50% higher than the profiling results. The increase in coverage is due to the directory cache conflicts and the discrete directory cache sizes. These results show that directory coverage and required directory size varies significantly with CPU scaling, *i.e.*, they scale sub-linearly with CPU scaling. Moreover the profiling framework in this thesis can be helpful to identify the minimum directory size.

8.2 DGD Technique Discussion

Section 4.3 also break down the entries by sharing degree, especially the ratio between private and shared directory entries. Section 6.3 shows this ratio changes

with CPU scaling, especially data cache scaling. In Section 6.3, Table 6.4 reports the percentage of shared entries over all live directory entries. Therefore, on average, the ratio of private entries over all directory entries decreases from 90.67% at 64KB private caches, to 72.01% at 1MB private caches and 60.85% at maximum PRD.

As discussed in Section 2.2, many directory techniques focus on *exploiting private data* to reduce directory cache size [20,22,23,45]. The profiling results show these techniques can be very effective, because the private entries still account for a majority of the directory entries. However, these techniques are also sensitive to CPU scaling because the portion of private entries is decreasing with cache size scaling.

The profiler's analyses can show how well these techniques will theoretically do. In particular, the curves labeled "$\geq$ 2 sharers" in Figure 6.3 show the coverage for shared entries. If a directory technique can reduce directory entries for private data, then in the limit, the "$\geq$ 2 sharers" curves in Figure 6.3 show the best that such a technique can ever do–*i.e.*, by eliminating *all* private directory entries. Figure 6.3 shows removing private entries can potentially lower coverage significantly. However, the reduction becomes smaller as CPUs scale due to increased sharing in the data cache.

To further illustrate this, the profiler is modified to compute the coverage for DGD technique [20] at 64 cores. DGD exploits the observation that private data tend to occur in large contiguous regions. DGD employs a "region entry" to track such contiguous private data. A "region entry" only occupies one directory entry but is able to track 64 consecutive blocks (4KB of memory) if they are all

accessed by the same core, *i.e.*, the region owner. When another core other than the region owner accesses the data in this region, the region is not private anymore. A "block entry" (for tracking individual cache blocks) is created for this data block. The "region entry" still exists to track the blocks that remain private to the region owner. Normal "block entries" and "region entries" are stored in the same structure. Each directory entry contains 64 bits, to either track 64 cores in a "block entry" using a full-map approach, or 64 consecutive blocks in a "region entry." The main advantage of DGD is its ability to reduce the number of entries required to track all cache blocks.

8.2.1 Experimental Setup

The experiment is done using a modified profiler, which is similar to the one explained in Chapter 5. Similar to Section 5.2.2, counters are maintained to track the number of live directory entries for the DGD technique.

Similar to Section 5.2.2, the profiler computes the average number of live DGD entries across time by accumulating the lifetime of all DGD directory entries and then averaging them by total time. A set of region entry counters, *one per 64 consecutive blocks (4KB of memory) contained in all of the LRU stacks for every capacity* are maintained, to track the number of data blocks and the region owner for this region. A set of region lifetime counters are also maintained along with the region entry counters, one per 4KB of memory for every capacity.

When a reference initiates a new region entry lifetime, its time is stored into

the region lifetime counter and the region owner is also recorded. When another data block in this region accessed by the region owner, the DGD entry counter increases by one. When another core other than the region owner accesses the data in this region, DGD entry counter decreases by one if the data is currently tracked by the region entry, because the block is not private any more.

When the lifetime of block for one cache capacity terminates, the profiler determines whether it is tracked by the region entry or the block entry. If the data block is tracked by the block entry, the profiler computes the lifetime of block, using directory lifetime counters explained in Chapter 5 and adds this value into an aggregated counter for this capacity. If the data block is tracked by the region entry, the profiler decreases region entry counter by one. When the counter reaches zero, indicating all the private blocks belongs to the owner in this region has exited the cache, the profiler computes the lifetime of the region entry, using region lifetime counter and add this value into an aggregated counter for this capacity. There is one aggregated counter provided for each capacity. At the end of the program, the values in the aggregated counters are divided by the total time to obtain the average number of DGD entries for each capacity.

8.2.2 Experiment results

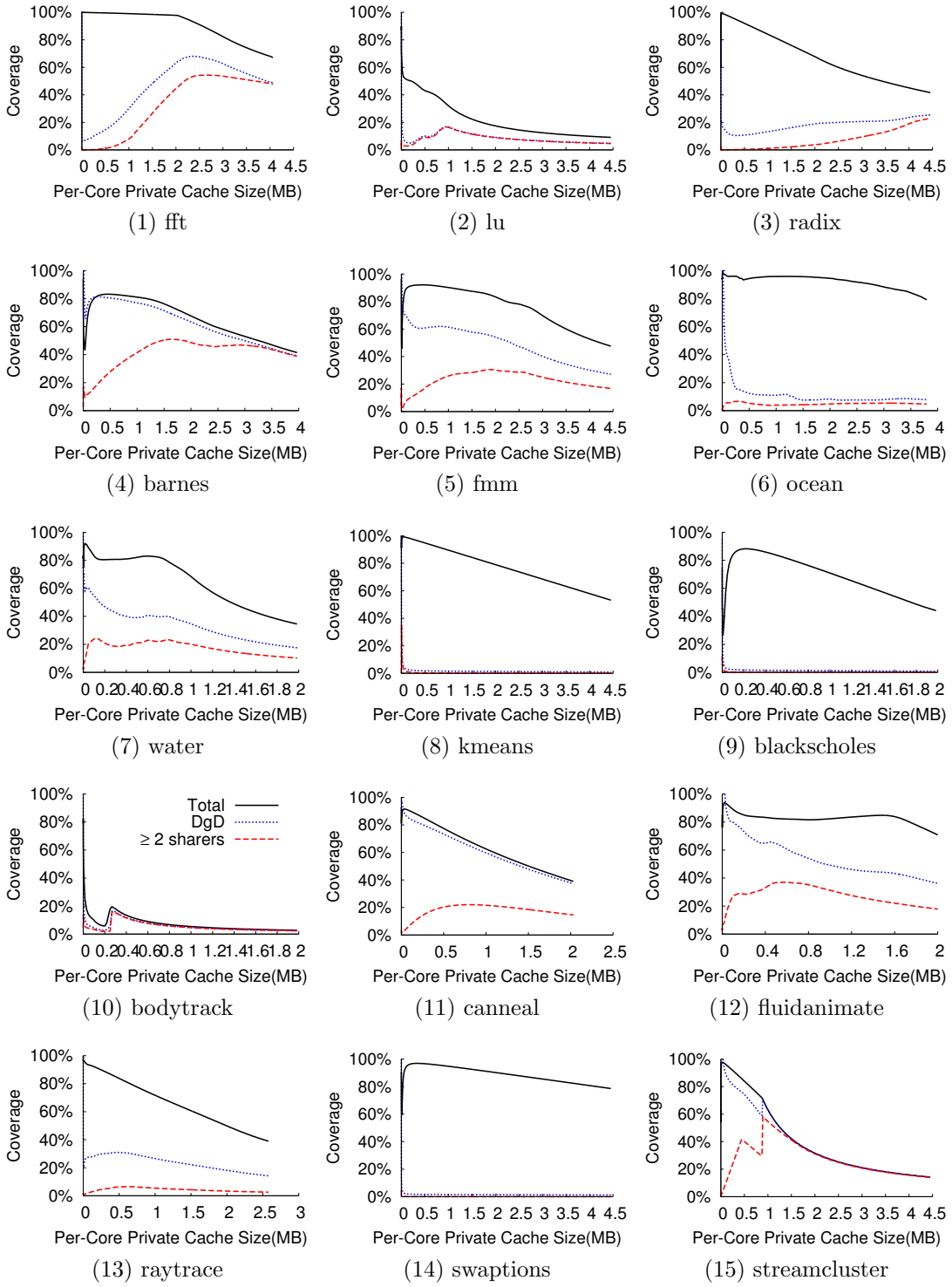
Figure 8.2 plots the coverage for DGD at 64 cores in the dotted lines, labeled as "DGD", along with the same "Total" and "$\geq$ 2 sharers" from Figure 6.3 for comparison. Notice the "DGD" lines always lie between the "Total" and "$\geq$ 2

sharers" lines. This is because the "$\geq$ 2 sharers" lines is the theoretical limit for techniques that exploit private data.

Figure 8.2 shows in most cases, coverage for the DGD technique drops with cache size scaling because the total coverage drops with cache size scaling. But in fft and radix, the coverage for DGD increases due to the increased sharing with data cache scaling. To quantify this phenomenon, the first three columns in Table 8.2 report the DGD coverage for private cache sizes of 256KB, 1MB and the maximum PRD. At 256KB, DGD coverage on average is 33.5%; at 1MB, DGD coverage on average is 30.8%; and at maximum PRD, DGD coverage on average is 20.0%. In most cases, Figure 8.2 shows the DGD coverage is less than 50%. Hence, the results show DGD is a good technique for achieving small directory cache size.

On the other hand, Figure 8.2 also shows the DGD technique's reduction in coverage over the total coverage is not as big when cache sizes scale. The next three columns in Table 8.2 report the DGD coverage reduction compared to the total coverage for private cache sizes of 256KB, 1MB and the maximum PRD. At 256KB, DGD's coverage reduction on average is 49.6%; at 1MB, DGD's coverage reduction on average is 42.7%; and at maximum PRD, DGD's coverage reduction on average is 26.7%. This shows the DGD coverage reduction goes down with cache size scaling, but the reduction is still significant, even at maximum PRD.

The DGD work does not consider the cache capacity scaling effect [20], but its predecessor, SCT [22], which proposed the idea of a "region entry" did study cache size scaling. The authors observed that SCT's reduction of the directory size compared to a 2x sparse directory increases with cache size scaling. Thus, the



(1) fft (2) lu (3) radix

(4) barnes (5) fmm (6) ocean

(7) water (8) kmeans (9) blackscholes

(10) bodytrack (11) canneal (12) fluidanimate

(13) raytrace (14) swaptions (15) streamcluster

Figure 8.2: DGD with 64 cores

Table 8.2: DGD Coverage; Compare DGD with profiled total coverage

Benchmark	DGD Coverage			DGD Coverage Drop		
	256KB	1MB	∞	256KB	1MB	∞
fft	10.0%	30.1%	48.8%	89.8%	69.0%	18.4%
lu	5.6%	16.7%	4.7%	43.8%	15.4%	4.4%
radix	10.8%	13.2%	25.5%	85.2%	70.8%	16.2%
barnes	81.3%	76.9%	39.1%	0.2%	4.1%	2.3%
fmm	62.2%	61.4%	27.1%	29.4%	28.8%	20.4%
ocean	15.7%	11.0%	7.8%	80.3%	84.8%	71.3%
water-spatial	44.5%	34.6%	26.2%	36.1%	33.8%	25.4%
kmeans	2.2%	1.6%	0.9%	95.2%	87.9%	52.3%
blackscholes	1.6%	1.2%	0.8%	86.5%	69.7%	49.2%
bodytrack	5.3%	15.9%	15.9%	12.7%	2.2%	2.2%
canneal	80.2%	59.7%	37.3%	5.4%	2.8%	1.7%
fluidanimate	69.6%	49.2%	36.2%	14.6%	33.1%	34.6%
raytrace	29.8%	26.5%	14.2%	60.1%	45.2%	24.8%
swaptions	1.8%	1.5%	1.2%	95.0%	93.2%	77.4%
streamcluster	81.5%	63.1%	14.2%	10.3%	0.0%	0.0%
Average	33.5%	30.8%	20.0%	49.6%	42.7%	26.7%

authors concluded that the opportunity increases with cache size scaling. However, this scaling result is a combination of two effects on reduction in of total coverage and the increase in multi-sharer entries.

To illustrate, when the data cache is small (at 256KB), the coverage for all directory entries is 78.0%, but 85.5% of the entries are private. DGD coverage is small because it can compress the private entries. When the data cache is larger (at maxPRD), only 60.8% of the entries are private, but the coverage for all directory entries is 40.3%. The DGD technique does not have as many private entries to compress, but the coverage is still small because the total coverage is low to begin with. Therefore, while techniques that can compress private data entries will still be useful when scaling the architecture, their opportunity actually decreases with scaling.

8.3 SCD Technique Discussion

Besides the relationship between private and shared entries, a final observation is that the sharing occurs non-uniformly in the data cache, as most entries only exhibit 2 or 3 sharers. As discussed in Section 6.3, Table 6.4 reports the percentage of entries for "≥4 sharers," showing that at 1MB, only 7.74% of the entries have more then 4 sharers. And, at maximum PRD, only 13.29% of the entries have more then 4 sharers.

As discussed in Section 2.2, many techniques take advantage of the fact that most directory entries are narrowly shared entries. They employ limited pointers to represent the sharer list instead of a full-map approach, and choose different techniques when the pointers overflow, such as software fallback [14], chained pointers [13] and extra entries [18]. However, these techniques are also sensitive to cache size scaling because the sharing increases.

To further illustrate this point, the profiler is modified to compute the coverage for the SCD technique at 256 cores [18]. SCD employs different techniques to handle directory entries with narrow sharing and wide sharing. For blocks with narrow sharing, SCD uses one entry with 3 pointers. When the pointers overflow, the entry becomes the root of a hierarchical directory entry with as many leaf entries as needed. Each leaf entry contains the leaf id and also a 16-bit part of the full-map bit vector. The main size reduction is from the size of the directory entry. Compared to a 256-bit full map entry, the SCD entry is 32 bits; therefore reducing the directory entry by 87.5%.

8.3.1 Experimental Setup

The experiment is done using a modified profiler, which is similar to the one explained in Chapter 5. Similar to Section 8.2.1, counters are maintained to track the number of live directory entries for the SCD technique.

Similar to Section 8.2.1, the profiler computes the average number of live SCD entries across time by accumulating the lifetime of all SCD directory entries and then averaging them by total time. A set of SCD entry counters, *one per unique data block contained in all of the LRU stacks for every capacity* are maintained, to track the number of live directory entries for this region. A set of SCD lifetime counters are also maintained along with the SCD entry counters, one per data block for every capacity.

The SCD entry counters count the number of live directory entries for this region. When the number of sharers is smaller than the number of available pointers, *i.e.*, 3 in this case, the SCD counter value is one. When the number of sharers exceed the number of available pointers, SCD counter value is computed based on the sharing pattern. In this implementation, each leaf directory entry tracks 16-bit part of the full-map bit vector. Therefore, if there is at least one sharer in the 16-bit part, a leaf entry is counted. Every time there is a change in value in SCD entry counter, the profiler computes the time interval by subtract the current time by the time stored in the SCD lifetime counter and updates the SCD lifetime counter. Then the profiler stores the product of SCD entry count times the time interval in to an aggregated counter. There is one aggregated counter provided for each capacity.

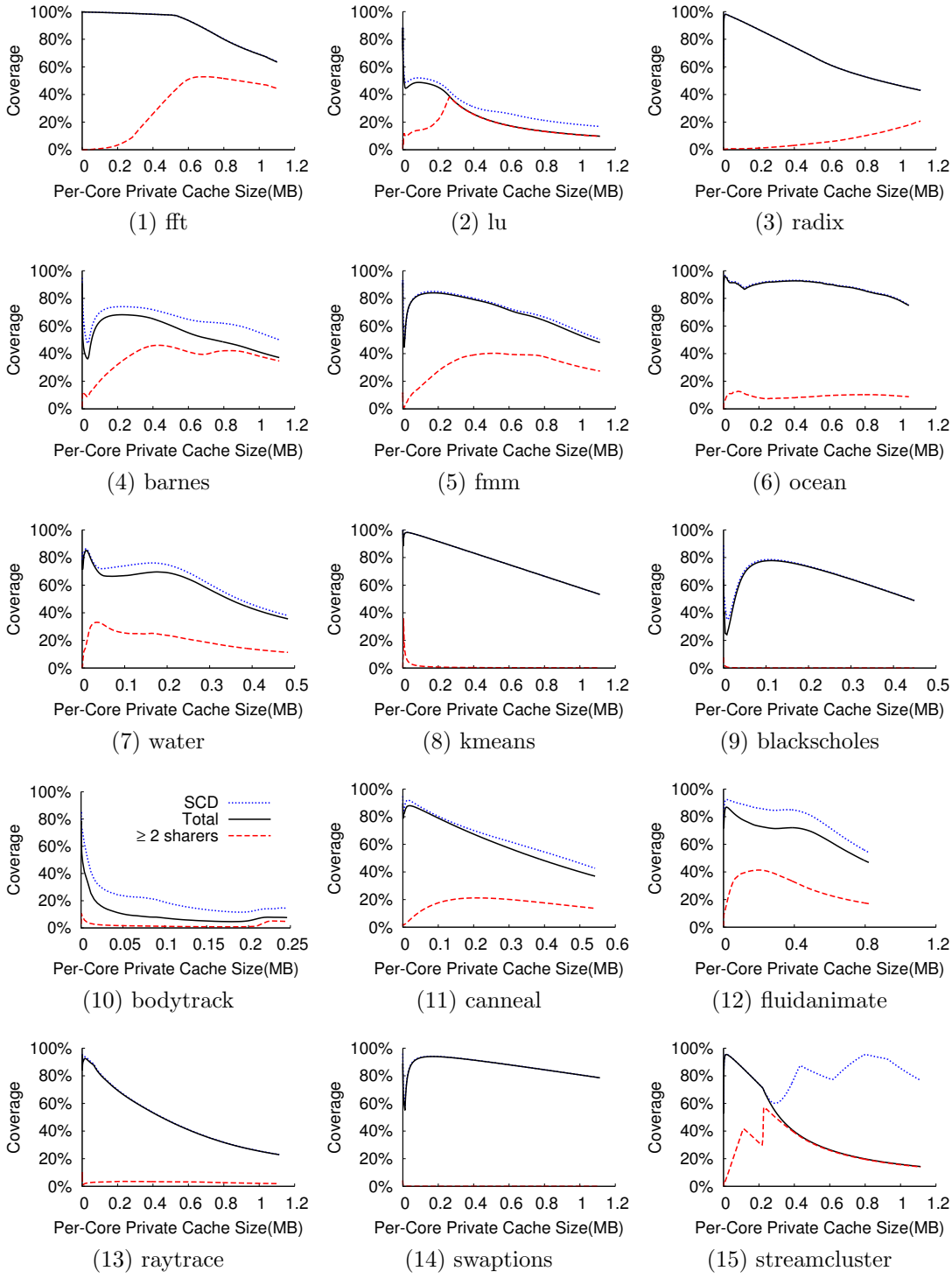
At the end of the program, these values are divided by the total time to obtain the average number of SCD entries for each capacity.

8.3.2 Experimental Results

Figure 8.3 plots the coverage for SCD at 256 cores in the dotted lines, labeled as "SCD", along with the same "Total" and "$\geq$ 2 sharers" from Figure 6.3 for comparison. Notice the "SCD" lines always lie above the "Total" lines because a single entry in a conventional non-hierarchical directory technique may require multiple entries in the SCD technique.

Figure 8.3 shows in most cases, the coverage for SCD still drops with cache size scaling because the total coverage drops with cache size scaling. But in streamcluter, due to its wide sharing and unique sharing pattern, SCD coverage increases significantly, comparing to the total coverage. To quantify this phenomenon, the first three columns in Table 8.3 report the SCD coverage for cache sizes of 256KB, 1MB and the maximum PRD. At 256KB, SCD coverage on average is 79.7%; at 1MB, SCD coverage on average is 72.9%; and at maximum PRD, SCD coverage on average is 48.6%. Although SCD directories require more entries than a full-map directory, the coverage still drops below 50% at the maximum PRD. Considering the directory size reduction SCD brings by reducing the sharer list size, SCD provides a significant reduction in the directory cache size.

Moreover, the increase of SCD's coverage is small. To illustrate, the next three columns in Table 8.3 report the SCD coverage increase for cache sizes of 256KB,



(1) fft (2) lu (3) radix

(4) barnes (5) fmm (6) ocean

(7) water (8) kmeans (9) blackscholes

(10) bodytrack (11) canneal (12) fluidanimate

(13) raytrace (14) swaptions (15) streamcluster

Figure 8.3: SCD with 256 cores

Table 8.3: SCD Coverage; Compare SCD with profiled total coverage

Benchmark	SCD Coverage			SCD Coverage Increase		
	256KB	1MB	∞	256KB	1MB	∞
fft	99.6%	98.9%	63.7%	0.0%	0.0%	0.3%
lu	51.7%	44.1%	16.9%	3.5%	3.6%	7.1%
radix	95.4%	83.8%	43.0%	0.1%	0.0%	0.1%
barnes	62.4%	74.0%	50.0%	8.0%	5.9%	12.9%
fmm	80.3%	84.0%	50.3%	0.5%	0.9%	2.4%
ocean	92.0%	92.2%	75.3%	0.7%	0.5%	0.5%
water-spatial	72.4%	69.1%	38.0%	6.0%	4.6%	2.4%
kmeans	97.2%	89.6%	53.3%	0.0%	0.0%	0.0%
blackscholes	74.7%	69.4%	49.0%	1.3%	0.3%	0.2%
bodytrack	22.8%	14.2%	14.4%	13.7%	6.6%	6.8%
canneal	85.2%	65.9%	42.7%	1.6%	3.8%	5.8%
fluidanimate	90.6%	85.1%	54.1%	8.6%	13.1%	7.3%
raytrace	88.9%	65.2%	23.0%	0.8%	0.4%	0.1%
swaptions	91.0%	93.6%	78.6%	0.5%	0.1%	0.0%
streamcluster	91.2%	64.4%	76.7%	0.0%	1.0%	62.4%
Average	79.7%	72.9%	48.6%	3.0%	2.7%	7.2%

1MB and the maximum PRD, compared to a full-map directory. At 256KB, SCD increases the coverage by 3.0% on average; at 1MB, SCD increases the coverage by 2.7% on average; and at the maximum PRD, SCD increases the coverage by 7.2%. This shows that the increased sharing at large cache sizes reduces SCD's advantage as cache size scales, but the impact is not significant.

8.4 Multi-Level Technique Discussion

Another characteristic of the directory is that directory entries do not get accessed uniformly. In fact, as discussed in Section 6.4, a small part of the directory receives a disproportionately large fraction of the directory accesses. As Table 6.7 shows, at 256KB, directory cache with 18.75% coverage receives 98.6% of the T2

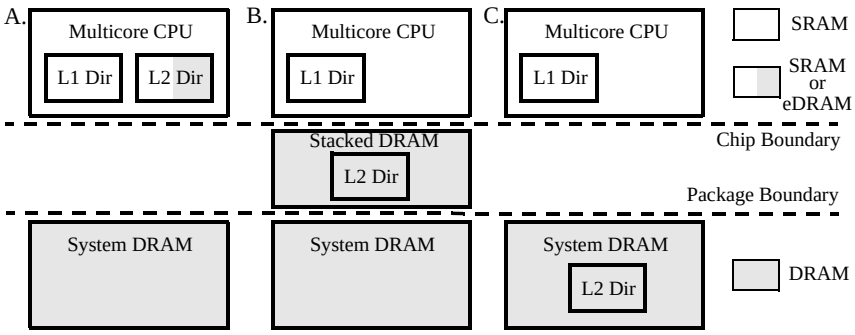


Figure 8.4: Multi-level directory cache implementations.

directory accesses and 37.6% of the total directory accesses; and at 1M, it receives 95.7% T2 directory accesses and 50.5% of the total directory accesses.

As discussed in Section 4.1, the reason to distinguish T1 and T2 transactions is the characteristic of latency tolerance. T2 transactions need to be performed before performing the operation to the corresponding data, such as checking the sharer list and then invalidating the sharers. These transactions are on the critical path. In contrast, T1 transactions only involve creating new entries, and the corresponding data are in the next level. Although directory access is still needed to determine the data is not in the private cache, this can be speculated, by performing the directory access and data access in parallel. Therefore, T1 transactions are tolerant of slower directory access.

As discussed in Section 2.2, recent directory designs have proposed asymmetric storage techniques to exploit this characteristics. They propose two levels of directory cache, a fast *L1 directory cache* backed by a slower *L2 directory cache* [21, 24]. In particular, directory entries that receive a lot of T2 accesses should be kept in the L1 directory cache, so that it can provide low latency to these entries with accesses

that are on the CPU's critical path. In contrast, directory entries solely involved T1 access can reside in the L2 directory cache, because T1 accesses are latency tolerant and a longer latency for T1 accesses will not affect the CPU performance.

To achieve high performance, the L1 directory cache should be implemented in SRAM on-chip, similar to a conventional directory cache. But there are different implementation for the L2 directory cache, as shown in Figure 8.4. First, the L2 can be implemented in on-chip SRAM with some energy reduction techniques that increase access latency to reduce the energy requirement for the directory. The L2 can also be implemented in eDRAM to reduce the area requirement for the directory. PS-Dir [21] used this approach.

Alternatively, the L2 directory cache can be implemented in off-chip DRAM, either as a stacked die on top of the CPU die or in main memory. WayPoint [24] implemented the L2 directory cache in system DRAM. Implementing L2 off-chip essentially provides unlimited capacity, but also increases the access latency and energy.

In this section, the cache simulator in Section 7.1 is modified to simulate a two-level directory cache, to understand directory locality further. The simulator models the implementation described in Figure 8.4(B). Using stacked DRAM is a tradeoff between the high area requirement in Figure 8.4(A) and the high latency in Figure 8.4(B).

Table 8.4: Data and directory cache parameters for two level directory.

Data Cache Hierarchy			
Private Only		Private + Shared	
Private L1	16KB (4-way)	Private L1	16KB (4-way)
Private L2	64KB (8-way)	Private L2	64KB, 128KB
Private L3	256KB, 512KB		256KB, 512KB (8-way)
	1M, 2M (8-way)	Shared L3	128MB (8-way)
Stacked DRAM L4	1G (8-way)	Stacked DRAM L4	1G (8-way)
Directory Cache Hierarchy			
L1: 18.75% Coverage of last level private cache(6-way) L2: In-cache directory with L4			

8.4.1 Experimental Setup

This simulator models two different on-chip cache hierarchies, with a stacked-DRAM cache, and a two-level directory cache. In one hierarchy, three levels of private cache are simulated, and in the other, two levels of private cache and a shared cache are simulated. Similar to the cache simulator in Section 7.1, all caches are inclusive and a MESI protocol is used to maintain private cache coherence. Four different last-level private cache sizes in both hierarchies are simulated to show the scaling effects. Table 8.4 lists the simulation parameters for the cache hierarchies.

The simulator also models a two-level directory cache. The L1 directory cache is 6-way set associative, implemented on chip and the L2 is an in-cache directory, implemented with the stacked-DRAM cache. The L1 directory is sized to have 18.75% coverage over the last-level private data cache, as shown in Table 8.4. And while the in-cache directory is large, it is still small compared to what stacked-DRAM can provide. Full-map implementation is used in these two directory caches

to eliminate the transactions related to imprecise sharer tracking.

The following describes the directory cache management policy. On a directory access, the L1 directory is checked first, and if the L1 misses, than the L2 directory in the stacked-DRAM is checked. On an L2 miss, a new directory entry is filled into the L1 directory cache along with the data. This access is always a T1 transaction and will start a new lifetime in the directory cache. This entry will then receive one or more T2 transaction hits (which are satisfied with low latency) during its time in the L1 directory cache. Later, the entry can be evicted from the L1 and written back to the L2 directory cache. On an L2 hit, the directory entry is promoted to the L1. This access can be a T1 transaction or a slow T2 transaction, depending on whether the associated data is in the private cache or not.

On the other hand, when the data associated with a directory entry is evicted in data cache, the entry in the L1 is notified to keep the sharer list updated (Both dirty block and clean block evictions notify the directory [19]). If there is no existing data block associated on chip according to the sharer list, the entry is invalidated to make room for other entries. However, the entry is not notified if it is in the L2 to reduce the traffic to the stacked-DRAM, though this will prolong the lifetime of the directory entry due to imprecise tracking. In addition, when a dirty block is evicted from the on-chip data cache, a data writeback is performed to the stacked-DRAM. In this case, an L2 directory update is performed at the same time to update the state to invalid.

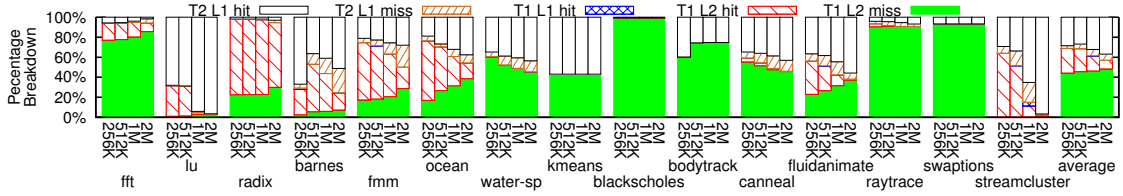


Figure 8.5: Hit and miss rates for T1 and T2 lookups at different levels of the directory cache for different private cache sizes.

8.4.2 Private Cache Results

This section looks at how multi-level directory caches can exploit temporal locality. Figure 8.5 breaks down the rate of L1 and L2 directory cache hits and misses in terms of T1 or T2 transactions as a percentage of total directory accesses. Figure 8.5 shows the results for four different cache sizes for each benchmark. Each bar provides the breakdown into five categories: T2 accesses that hit or miss in the L1 directory cache, T1 accesses that hit in either the L1 or L2 directory cache, and T1 accesses that miss in both directory caches.

Figure 8.5 shows the majority of T2 transactions hit in the L1 directory cache. As the "T2 L1 Hit" and "T2 L1 Miss" components in Figure 8.5 show, the T2 hit rates across different cache sizes are 27–37% of all directory transactions, but are 83–91% of all T2 transactions. The simulations confirm that a small L1 directory cache can exploit temporal reuse across T2 transactions, allowing the most latency-critical accesses (T2) to be serviced rapidly. Only 14% of the T2 accesses on average incur longer L2 accesses.

Compared to the locality results in Figure 6.6 and Table 6.7, the cache simulation results are different in a few ways. First, data cache conflicts and directory

conflicts are taken into consideration. Second, the profiler counts the T2 access to the entries with most accesses they receive in their lifetime, while the simulator uses a LRU replacement policy. Therefore, the simulation results have lower T2 hit rates. However, comparing Figure 8.1 and Figure 8.5 still shows the T2 hit rates are correlated to the profiling results.

Figure 8.5 also shows that the majority of the directory accesses miss in both directory caches. These accesses, labeled "T1 L2 Miss," are the T1 directory accesses that create new entries in the directory. They account for 44–48% of all directory accesses. Moreover, a small number of T1 accesses hit in the L1 directory cache. This is because some directory entries linger in the directory after their data blocks are evicted from the data cache. These accesses, labeled "T1 L1 Hit," account for only 0.01–0.2% of all directory accesses.

Lastly, Figure 8.5 shows T1 speculation can be very accurate. As discussed above, directory accesses and data accesses can be performed in parallel for T1 transactions because they only involve creating new entries. It is possible to speculate all L1 directory cache misses are T1 transactions, and fetch the directory entry and data at the same time. On average, only 7.5% of the L1 directory misses are T2 transactions; therefore, the speculation will be wrong only 7.5% of the time on average. When the speculation is wrong, a data block is fetched from the next level of the memory hierarchy needlessly. But because the L2 directory is implemented as an in-cache directory within the L4 data cache, this will usually not incur an extra main memory access. However, it does consume extra L4 cache bandwidth. Moreover, for all other L1 directory misses, L2 directory accesses with their data

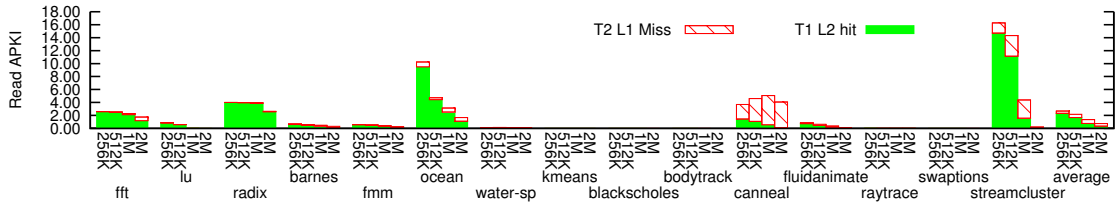


Figure 8.6: L2 directory cache read APKI for different private cache sizes.

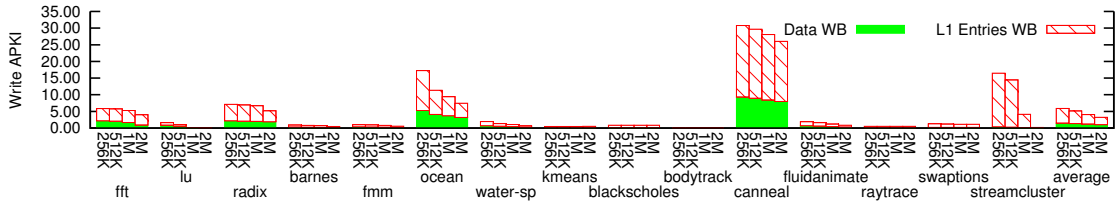


Figure 8.7: L2 directory cache write APKI for different private cache sizes.

accesses are overlapped.

Figures 8.6 and 8.7 report traffic to the L2 directory cache. When transaction can not be serviced from the L1 directory cache ("T2 L1 miss," "T1 L2 hit," and "T1 L2 miss"), the next level stacked-DRAM cache along with its in-cache directory is accessed. The "T2 L1 miss" components cause extra accesses and bandwidth to the stacked-DRAM cache because the data is in the private cache on chip. On the other hand, the "T1 L2 hit" components do not cause extra accesses to the stacked-DRAM cache because the accesses need to fetch the data anyway, but they do consume extra bandwidth to fetch the directory entry to make sure the data is not in the private cache. In addition, the "T1 L2 miss" components do not cause extra accesses nor bandwidth to the stacked-DRAM because it is a stacked-DRAM cache miss.

Figures 8.7 shows there are two types of write transactions to the directory. One is due to the L1 directory cache writeback. This transaction causes extra

accesses and bandwidth to the stacked-DRAM. The other is the private data cache writeback. Because only dirty data is written back and updates the directory along with it, this transaction will not cause extra access (because dirty data writeback is necessary), but will cause extra bandwidth to the directory.

Figure 8.6 reports the read traffic (APKI) to the L2 directory, including "T2 L1 miss" and "T1 L2 hit." Figure 8.6 shows the read traffic to L2 directory is highly benchmark dependent. 10 out of 15 benchmarks have very little traffic, less then 1 APKI. The APKI for the remaining benchmarks, fft, radix, ocean, canneal and streamcluster, are between 0.2–16.3 APKI, which is significant. However, the majority of the read traffic is latency tolerant T1 transactions. 13 out of 15 benchmarks have less than 0.8 APKI read traffic caused by "T2 L1 miss." The APKI of "T2 L1 miss" for canneal and streamcluster varies from 0.1 to 4.5. But on average, the APKI for "T2 L1 miss" is between 0.3–0.6 and the APKI for "T1 L2 hit" is between 0.4–2.3, which is not significant for stacked-DRAM.

Figure 8.7 reports the write traffic (APKI) to the directory, including "Data WB" and "L1 entry WB." Similar to Figure 8.6, Figure 8.7 shows the write traffic to the L2 directory is also very benchmark dependent. 10 out of 15 benchmarks have less than 2 APKI. The write APKI for fft, radix, ocean and streamcluster is between 0.1–16.2 APKI and the write APKI for canneal is between 18.0–21.4 APKI. However, *all* of the write traffic can be buffered and pipelined to be performed without stalling the CPU. Though write traffic is significant in a few benchmarks, on average, the APKI for "Data WB" is between 1.0–1.5 and APKI for "L1 entry WB" is between 2.2–4.4, which is not significant for stacked-DRAM.

8.4.3 Shared Cache Impact

As discussed in Section 4.1, directory cache behavior is determined by private caches. However, when considering the latency criticality of directory accesses, the shared cache also comes into play because it changes the sharing point. In the previous section, only private caches existed on chip; therefore, the sharing point is off-chip, as indicated in Figure 4.1. The data accesses associated with T1 transactions happen off-chip, thus the T1 transactions are latency tolerant. However, the shared cache moves the sharing point on chip The data for certain T1 transactions may now be found in the shared cache on chip. In that case, the directory cache accesses are on the CPU's critical path and become latency critical.

On the other hand, shared caches also provide a potential optimization: they overlap the directory cache accesses with the data accesses if *the directory entry update is deferred*. Usually, when a local cache read miss happens, the directory cache has to be checked to determine the state and the location of the cache block before the data can be fetched. However, the shared cache may already contains the data, and can also include the state information with its corresponding cache block. In this case, if the access is a data read and the state of the cache block is shared or invalid, indicating the shared cache holds the most recent copy, then the shared cache data can be forwarded to the requesting core immediately without further stalling the core. The directory cache still needs to be updated, such as creating a new entry or adding a sharer to the sharer list, but this can happen off the critical path of the core's data access.

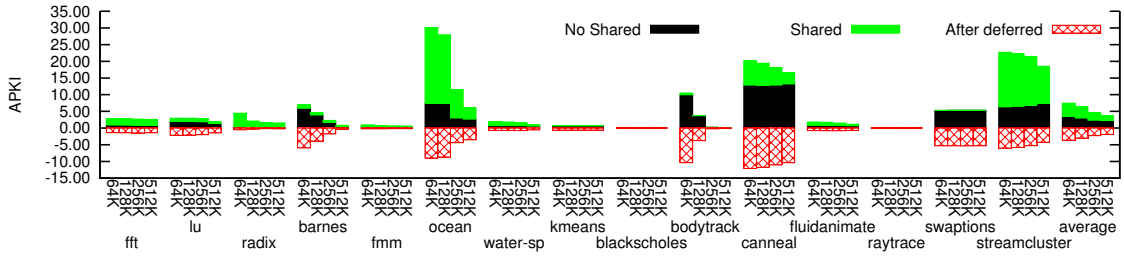


Figure 8.8: Latency critical L2 directory cache access APKI with or without shared caches and deferred

Figure 8.8 shows how a shared cache affects the latency tolerance of the L2 directory accesses. The "No shared" components in Figure 8.8 reports the "T2 L1 miss" components in the private + shared cache hierarchy from Table 8.4. The "Shared" components report the extra latency critical directory accesses, *e.g.*, the T1 directory transactions that hit in the shared cache. The "After deferred" components report the latency critical directory accesses that cannot be deferred, *e.g.*, the directory accesses that cannot be overlapped with their data accesses due to inconsistent states.

Averaged across all benchmarks, Figure 8.8 shows the latency critical portion of the directory accesses almost double, increasing from 2.2–3.5 APKI to 3.7–7.4 APKI. This is because a large portion of "T1 L2 hit" become latency critical due to the shared data cache. Due to good temporal reuse, T1 transactions have a high possibility of hitting in the shared cache.

Although shared cache hits eliminate one source of latency tolerance, it also provides a chance for optimization, as discussed above. Figure 8.8 shows after deferring some directory accesses, the latency critical APKI drops to 1.9–3.7, even lower than the original "T2 L1 miss" component in some cases. This is because not

only can the directory entry update for T1 transactions be deferred, but it works for read forwarding T2 transactions as well (Transaction 9 and 10 in Table 4.1). In these cases, the requesting core does not have to wait for the directory access to determine which remote core has the data because the most recent data copy is in the shared cache.

8.5 Directory Access Frequency Discussion

It is well known that cache misses reduce when scaling cache size, and directory accesses are closely related to cache misses as explained in Section 4.1. The analyses in Section 4.2 quantify this phenomenon and breakdown its components. This can help architects make design tradeoffs in directory caches as CPUs scale.

The first observation is that the total cache-miss induced directory accesses drops with CPU scaling, especially cache scaling as shown in Figure 6.1. As discussed in Section 2.2, many directory techniques proposed to reduce directory size do so at the expense of increased directory access latency. For example, some techniques employ complex hashing functions, such as Cuckoo Directory [19], SCD [18], and Tagless directories [46]. Some other techniques require multiple access to the directory, such as hierarchical directories [16, 17], software fallback [14] and chained pointers [13]. Because the directory accesses drop as CPUs scale, they make up a smaller fraction of the overall execution time. Therefore, the results from Section 6.2 imply that trading off higher access latency to reduce directory size is a good idea as CPUs scale.

The second observation is that the percentage of T2 transactions over total cache-miss induced directory accesses(T1+T2) increases with CPU scaling. As discussed in Section 2.2, some directory techniques employ re-insertion techniques to reduce the directory conflicts, such as Cuckoo Directory [19] and SCD [18]. These techniques increase the cost of directory accesses unevenly. In particular, the insertion of new directory entries (T1 translations) is more expensive in these techniques. Though T1 accesses constitute the majority of the directory accesses in small caches, they become less significant when cache size scales. So, the performance penalty for these techniques will be less with CPUs scales. Therefore, the results imply that trading off higher insertion latency to smaller directory size is also a good idea as CPUs scale.

Chapter 9: Conclusion

Reuse distance is a useful tool to study the locality in the data cache, for both sequential and parallel benchmarks. This thesis extends the reuse distance analysis to study the directory cache behavior because directory is one of the main bottlenecks in multicore processor scaling. It proposes the relative reuse distance between sharers to analyze the directory and extract insights on directory's architecture dependency.

This thesis builds a profiler using PIN-tool based on reuse distance analysis to study how directory access frequency, directory content information and directory access distribution changes with core count and cache size scaling. In terms of access frequency, the profiling results show directory accesses drop significantly with data cache size scaling. In terms of directory content, the profiling results show the directory coverages also drop significantly with data cache size scaling. In terms of access distribution, the profiling results show the there is locality in the directory. The profiling results also show that cache size scaling has a much bigger effect on directory than core count scaling.

Cache simulation studies is done to validate the profiling results. The validation results shows that the profiler is accurate enough to provide directory behavior

insights. This thesis also does case studies on four representative directory techniques using the insights from the profiler, including Cuckoo, DGD, SCD and multi-level techniques, and quantifies how multicore scaling will impact on them. The case study on Cuckoo technique shows that the required directory size scales sub-linearly with CPU scaling. The case study on DGD technique shows that the opportunity of compressing private data decreases with CPU scaling. The case study on SCD technique shows that reducing sharer list size is an effective technique with CPU scaling. And the case study on multi-level technique shows splitting directory into multiple level can be a promising technique.

In conclusion, this thesis provides a tool for architects to study the whole design space of directory behavior in relatively short time, providing insights for designing the directory.

Bibliography

[1] Anant Agarwal, Liewei Bao, John Brown, Bruce Edwards, Matt Mattina, Chyi-Chang Miao, Carl Ramey, and David Wentzlaff. Tile Processor: Embedded Multicore for Networking and Multimedia. In *Proceedings of the Symposium on High Performance Chips*, 2007.

[2] Yatin Hoskote, Sriram Vangal, Nitin Borkar, and Shekhar Borkar. Teraflop Prototype Processor with 80 Cores. In *Proceedings of the Symposium on High Performance Chips*, 2007.

[3] Nebojsa Novakovic. Intels xeon phi knights series expands in 2015. November 2013.

[4] Lisa Hsu, Ravi Iyer, Srihari Makineni, Steve Reinhardt, and Donald Newell. Exploring the Cache Design Space for Large Scale CMPs. *ACM SIGARCH Computer Architecture News*, 33, 2005.

[5] Li Zhao, Ravi Iyer, Srihari Makineni, Jaideep Moses, Ramesh Illikkal, and Donald Newell. Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*, 2007.

[6] H. S P Wong, S. Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, B. Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[7] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pages 459–462, 2005.

[8] H. Akinaga and H. Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[9] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, Vancouver, Canada, June 2000.

[10] Anoop Gupta, Wolf dietrich Weber, and Todd Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *In International Conference on Parallel Processing*, pages 312–321, 1990.

[11] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, Los Alamitos, CA, 1988.

[12] Jong Hyuk Choi and Kyu Ho Park. Segment Directory Enhancing the Limited Directory Cache Coherence Schemes. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, Washington, D.C., 1999.

[13] Guoying Chen. SLiD–A Cost-Effective and Scalable Limited-Directory Scheme for Cache Coherence. In *Proceedings of the Parallel Architectures and Languages Europe*, Heidelberg, Germany, 1993.

[14] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, 1991.

[15] Hongzhou Zhao, Arrvindh Shriraman, and Hya Dwarkadas. Space: Sharing pattern-based directory coherence for multicore scalability. In *In 19th Intl Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[16] Song-Liu Guo, Hai-Xia Wang, Yi-Bo Xue, Chong-Min Li, and Dong-Sheng Wang. Hierarchical Cache Directory for CMP. *Journal of Computer Science and Technology*, 25(2):246–256, March 2010.

[17] Deborah A. Wallach. PHD: A Hierarchical Cache Coherent Protocol (Master's Thesis). 1993.

[18] Daniel Sanchez and Christos Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, 2012.

[19] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 169–180, 2011. selected by the program committee for Best Student Papers session.

[20] Jason Zebchuk, Babak Falsafi, and Andreas Moshovos. Multi-Grain Coherence Directories. In *Proceedings of the 46th Annual International Symposium on Microarchitecture*, Davis, CA, December 2013.

[21] Joan J. Valls, Alberto Ros, Julio Sahuquillo, María E. Gómez, and José Duato. Ps-dir: a scalable two-level directory cache. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 451–452, New York, NY, USA, 2012. ACM.

[22] Mohammad Alisafaee. Spatiotemporal Coherence Tracking. In *Proceedings of the 45th Annual International Symposium on Microarchitecture*, December 2012.

[23] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 93–104, New York, NY, USA, 2011. ACM.

[24] John H. Kelm, Matthew R. Johnson, Steven S. Lumettta, and Sanjay J. Patel. Waypoint: Scaling coherence to thousand-core architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[25] Chen Ding and Trishul Chilimbi. A Composable Model for Analyzing Locality of Multi-threaded Programs. Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[26] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors? In *Proceeding of Compiler Construction*, 2010.

[27] Derek L. Schuff, Benjamin S. Parsons, and Jivay S. Pai. Multicore-Aware Reuse Distance Analysis. Technical Report TR-ECE-09-08, Purdue University, 2009.

[28] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[29] Meng-Ju Wu and Donald Yeung. Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs. *ACM Transactions on Computer Systems*, 31(1), 2013.

[30] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. Studying Multicore Processor Scaling via Reuse Distance Analysis. In *Proceeding of the International Symposium on Computer Architecture*, Tel-Aviv, Israel, June 2013.

[31] Meng-Ju Wu and Donald Yeung. Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs. In *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*, Galveston Island, TX, October 2011.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[34] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[35] Jichuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proc. of the 33rd Int'l Symp. on Comp. Arch.*, June 2006.

[36] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[37] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[38] Manuel E. Acacio, Jose Gonzalez, Jose M. Garcia, and Jose Duato. A New Scalable Directory Architecture for Large-Scale Multiprocessors. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, Washington, D.C., 2001.

[39] D.B. Gustavson. The scalable coherent interface and related standards projects. *Micro, IEEE*, 12(1):10–22, 1992.

[40] H. Nilsson and P. Stenstrom. The scalable tree protocol-a cache coherence approach for large-scale multiprocessors. In *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*, pages 498–506, 1992.

[41] Y. Chang and L.N. Bhuyan. An efficient hybrid cache coherence protocol for shared memory multiprocessors. In *Parallel Processing, 1996. Vol.3. Software., Proceedings of the 1996 International Conference on*, volume 1, pages 172–179 vol.1, 1996.

[42] Quing Yang, G. Thangadurai, and L.N. Bhuyan. Design of an adaptive cache coherence protocol for large scale multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 3(3):281–293, 1992.

[43] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Journal of Algorithms*, 2001.

[44] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 187–198, Washington, DC, USA, 2010. IEEE Computer Society.

[45] Blas Cuesta, Alberto Ros, Maria E. Gomez, Antonio Robles, and Jose Duato. Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks. *IEEE Transactions on Computers*, 62(3):482–495, 2013.

[46] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. A Tagless Coherence Directory. In *Proceedings of the 42nd International Symposium on Microarchitecture*, New York, NY, 2009.

[47] Yutao Zhong, Steven G. Dropsho, and Chen Ding. Miss Rate Prediction across All Program Inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[48] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[49] Collin McCurdy and Charles Fischer. Using pin as a memory reference generator for multiprocessor simulation. *ACM SIGARCH Computer Architecture News*, 33, 2005.

[50] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[51] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokham Memik, and Alok Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proceedings of the International Symposium on Workload Characterization*, 2006.

[52] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.