A

Using Multicore Reuse Distance to Study Coherence Directories

MINSHU ZHAO, The MathWorks, Inc.
DONALD YEUNG, University of Maryland at College Park

Researchers have proposed numerous techniques to improve the scalability of coherence directories. The effectiveness of these techniques not only depends on application behavior, but also on the CPU's configuration, *e.g.* its core count and cache size. As CPUs continue to scale, it is essential to explore the directory's application *and* architecture dependences. However, this is challenging given the slow speed of simulators. While it is common practice to simulate different applications, previous research on directory designs have explored only a few–and in most cases, only one–CPU configuration, which can lead to an incomplete and inaccurate view of the directory's behavior.

This article proposes to use *multicore reuse distance analysis* to study coherence directories. We develop a framework to extract the directory access stream from parallel LRU stacks, enabling rapid analysis of the directory's accesses and contents across both core count and cache size scaling. A key part of our framework is the notion of *relative reuse distance between sharers*, which defines sharing in a capacity-dependent fashion and facilitates our analyses along the data cache size dimension.

We implement our framework in a profiler, and then apply it to gain insights into the impact of multicore CPU scaling on directory behavior. Our profiling results show that directory accesses reduce by 3.3x when scaling the data cache size from 16 KB to 1 MB, despite an increase in sharing-based directory accesses. We also show that increased sharing caused by data cache scaling allows the portion of on-chip memory occupied by the directory to be reduced by 43.3%, compared to a reduction of only 2.6% when scaling the number of cores. And, we show certain directory entries exhibit high temporal reuse. In addition to gaining insights, we also validate our profile-based results, and find they are within 2-10% of cache simulations on average, across different validation experiments. Finally, we conduct four case studies that illustrate our insights on existing directory techniques. In particular, we demonstrate our directory occupancy insights on a Cuckoo directory; we apply our sharing insights to provide bounds on the size of SCD and DGD directories; and, we demonstrate our directory entry reuse insights on a multi-level directory design.

1. INTRODUCTION

The trend for multicore CPUs is towards integrating an increasing number of cores on chip. Today, energy-efficient CPUs, such as Intel's Phi [Intel 2014] and Tilera's Tile processors [Agarwal et al. 2007], already implement 10s of cores on a single die. In the future, processors with 100s of cores, *i.e.* large-scale chip multiprocessors [Hsu et al. 2005; Zhao et al. 2007], will be possible.

To enable scaling, architects have investigated directory-based cache coherence protocols. An important factor in the scalability of these protocols is the design of their coherence directories. Duplicate tag directories [Barroso et al. 2000] become impractical as CPUs scale because they require high associativity–*i.e.*, linear with the number of cores. In contrast, sparse directories [Gupta et al. 1990] maintain an explicit sharer list per cache tag which can be stored in arrays with low associativity, so they are more scalable. The main problem with sparse directories is capacity. Because both sharer lists and cache tags tend to increase with core count, the directory size can grow super-linearly. Worse yet, sparse directories also incur conflicts that cause directory-induced invalidations. So, they require over-provisioning to keep conflicts at a minimum.

Researchers have investigated numerous techniques to improve the capacity scaling of directories. One approach is to *reduce the sharer lists*. For example, sharers can be

This research was supported in part by NSF grant #CCF-1117042, and in part by DARPA grant #HR0011-13-2-0005. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

tracked imprecisely using limited pointers [Agarwal et al. 1988; Chaiken et al. 1991; Chen 1993; Choi and Park 1999], coarse vectors [Gupta et al. 1990], multilayer clustering [Acacio et al. 2001], or tagless arrays [Zebchuk et al. 2009]. Also, the directory can be implemented hierarchically [Guo et al. 2010; Wallach 1993], resulting in logarithmic sharer list growth. More recently, SCD [Sanchez and Kozyrakis 2012] combines limited pointers with hierarchical lists to compactly encode both narrow and wide sharing patterns. In addition to sharer list minimization, researchers have also tried to *reduce conflicts*. For example, Cuckoo [Ferdman et al. 2011] and SCD use multiple hash functions and iterative re-insertion to increase associativity by mapping entries to multiple non-conflicting ways.

Another important approach for improving the capacity scaling of directories is *exploiting private data*. Cuesta's work [Cuesta et al. 2011; Cuesta et al. 2013] uses TLBs to detect pages that are accessed by a single core, and omits tracking their cache blocks in the directory. PS-Dir [Valls et al. 2012] devotes a separate directory to track private data using minimally-sized (*i.e.* single pointer) sharer lists. And, SCT [Alisafaee 2012], MGD, and DGD [Zebchuk et al. 2013] recognize that private data tend to occur in large contiguous regions. Hence, they coalesce consecutive private cache blocks that are accessed by the same core, and track them as a single coherence unit.

Finally, instead of reducing the directory's footprint, yet another approach is to implement directories in *asymmetric storage*. For example, PS-Dir [Valls et al. 2012] provides a fast directory in SRAM for frequently accessed directory entries, and a slow directory in denser eDRAM for infrequently accessed entries. Also, WayPoint [Kelm et al. 2010] evicts infrequently accessed entries that do not fit in the on-chip directory to off-chip DRAM.

The effectiveness of these previous techniques depends in large part on how applications exercise the directory. One crucial factor is programs' *sharing patterns*–*e.g.*, the degree of sharing across different cache blocks, as well as the type of sharing–read *vs.* write. Another important factor is programs' impact on *directory access patterns*, including access frequency and distribution over different directory entries.

But in addition to applications, directory techniques are also highly sensitive to architecture–*i.e.*, to the CPU's configuration. For instance, varying core count will clearly affect the amount and frequency of sharing, and hence, the directory's behavior. But also, varying the data cache hierarchy can have a significant impact as well. This is because the directory's access stream is defined by data cache misses, so changing the caches–*e.g.*, scaling capacity–will change the directory's behavior. Specifically, *it can alter the perceived sharing patterns*. While sharing is inherently an application-level behavior, whether or not a program's sharing patterns manifest themselves in the data caches–and hence, become visible to the directory–in fact depends on the data cache size.

To understand the efficacy of directory techniques, it is essential to explore their sensitivity to both application and architecture. Unfortunately, this is challenging with existing methodologies. The problem is architectural simulation–the primary method for evaluating directory techniques–is extremely slow when modeling large core counts. Moreover, each simulation only evaluates one configuration, so fully exploring application- and architecture-specific behaviors requires running simulation sweeps. Given finite simulation bandwidth, researchers often limit the number of configurations explored. While it is common practice to vary applications (*i.e.*, using entire benchmark suites), architectural scaling is usually neglected. Among the myriad directory studies mentioned above, only a few have simulated different core counts or cache sizes [Alisafaee 2012; Ferdman et al. 2011; Kelm et al. 2010]. And even in those cases, only a small number of configurations were explored.

Recently, there has been significant interest in evaluating multicore cache hierarchies via *reuse distance (RD) profiles* [Berg et al. 2006; Ding and Chilimbi 2009; Jiang et al. 2010; Schuff et al. 2009; Schuff et al. 2010; Eklov et al. 2011; Wu and Yeung 2013; Wu et al. 2013]. A program's RD profile is its memory reuse distance histogram, capturing memory reference locality that determines the program's cache performance. In recent work, researchers have extended uniprocessor profiling to handle multicore CPUs by modeling *inter-thread interactions*. For example, private-stack reuse distance (PRD) profiling [Schuff et al. 2009; Schuff et al. 2010; Wu and Yeung 2013] uses per-thread coherent LRU stacks to model the interactions that occur across private data caches. Unfortunately, whereas uniprocessor profiles are architecture independent, multicore RD profiles are sensitive to reference interleaving. So, strictly speaking, PRD profiles are dependent on architecture (*i.e.*, cache size). But researchers have shown that for programs with *homogeneous threads*, changes in interleaving at different cache sizes are benign [Jiang et al. 2010; Wu and Yeung 2013]. This makes PRD profiles “pseudo” architecture independent, and thus accurate across cache size scaling. Research has also shown that PRD profiles are highly predictable across core count scaling as well [Wu and Yeung 2013]. So, a few profiles can analyze caching behavior across a large number of CPU configurations without having to simulate all of them.

In this article, we apply multicore RD analysis to study coherence directories. Our goal is to enable for directories the powerful analyses that reuse distance has already demonstrated for multicore data caches. To accomplish our goal, we develop a framework for extracting the directory access stream from PRD stacks, enabling analysis of the directory's access patterns and their impact on the directory's contents. A key notion in our framework is *relative reuse distance between sharers*. In particular, the relative reuse distance between shared blocks within PRD stacks determines the cache sizes for which the sharing will manifest itself, permitting analysis of the directory's contents at different cache sizes. Due to the cache-size independence of PRD for programs with homogeneous threads, we can perform such analyses at every possible private data cache size from a single PRD profile. Also, using existing insights on PRD profiles [Wu et al. 2013], we can analyze the directory's behavior across core count scaling too. This allows our framework to provide deep insights into how the directory's accesses and contents vary across different architecture configurations, and hence, the efficacy of different directory techniques as CPUs scale.

We implement our analyses in a PIN-based profiler [Luk et al. 2005], and use it to study directory behavior. Although our profiler affords the aforementioned benefits, it also has several limitations. Similar to previous multicore RD-based techniques, our approach is limited to programs with homogeneous threads in which the architecture independence assumptions hold true. In addition, our study only analyzes the user-level data access stream performed by the CPU's cores; we do not currently profile the instruction stream, nor do we account for system-level memory accesses such as those performed by the operating system or by external I/O devices via DMA. We also do not include in our profiles the additional memory accesses that would be performed by hardware prefetchers. Moreover, we only profile one application at a time, so we do not account for multiprogramming. And, unlike simulators, our profiler cannot model timing. So, while we can analyze data cache hits / misses and directory cache accesses and contents, we cannot analyze their impact on end-to-end performance.

Using our profiler, we obtain several insights into directory cache behavior. We show that a 3.3x drop in directory accesses occurs when data cache size scales from 16 KB to 1 MB, despite an increase in sharing-based directory accesses. We also find elevated sharing reduces the number of active directory entries needed to track all the sharers. In particular, the portion of on-chip memory needed to store the active directory entries decreases by 43.3% on average across data cache size scaling. This shows that for

certain applications, the directory's portion of the on-chip memory can be decreased significantly for large data caches. It also suggests techniques that can configure the directory on a per-program basis are worth exploring. In contrast to cache size scaling, we find core count scaling at a fixed capacity only increases the directory's accesses by 38.5% and decreases the portion of active directory entries by 2.6%, despite a 16x increase in core count. Lastly, our profiler shows accesses are distributed non-uniformly across directory entries. We find that 23.0% of the directory entries can account for 42.7% of all directory accesses and 83.1% of sharing-based accesses.

To validate our profiling results, we compare them against cache simulations. Our validation experiments show the profiled directory access counts are within 7.1% of simulation across cache size scaling on average, and 9.9% across core count scaling. Moreover, the profiled directory sizes are within 2.9% of simulation across cache size scaling on average, and 3.7% across core count scaling. We also validate the profiled number of directory entries that receive the majority of sharing-based directory accesses, and find they are within 9.7% of simulation across cache size scaling on average, and 8.6% across core count scaling.

Finally, we conduct four case studies to explore the implications of our profile-based insights on existing directory techniques. First, we show that for a real directory, *e.g.* Cuckoo, the portion of cache blocks with active directory entries does in fact decrease significantly as data cache size scales, dropping to only 37.5–87.5% of the total cache blocks in the private data caches. Second, we show our profile results provide a lower bound on the number of directory entries in an SCD directory. While many benchmarks achieve this lower bound, some benchmarks allocate noticeably more directory entries. Third, we show our profile results also provide a lower bound on the amount of coalescing that a DGD directory can achieve. At small data cache sizes, most cache blocks are private, so there is ample opportunity for coalescing. But at large cache sizes, many private cache blocks become shared, so the opportunity for coalescing diminishes and DGD is unable to achieve the lower bound for many benchmarks. And fourth, we show that a majority of sharing-based directory accesses hit in the L1 directory of a multi-level directory cache design, thus demonstrating the implications of our profile-based access distribution results.

The rest of this article is organized as follows. Section 2 discusses the directory behavior we analyze. Then, Section 3 presents our analysis framework, and Section 4 implements it in a profiling tool. Next, Section 5 reports our profiling results, and Section 6 validates them. Section 7 conducts our case studies for existing directory techniques. Finally, Section 8 covers related work and Section 9 concludes the article.

2. DIRECTORY ACCESSES AND CONTENTS

Figure 1 illustrates the on-chip cache hierarchy of a typical multicore CPU. At the top of the hierarchy are the cores and their private data caches, with multiple levels of private cache per core (only the last level is shown in the figure). Below the private caches is the CPU's *sharing point* where the directory sits, which is labeled "Directory Cache." Optionally, there may also be a shared data cache at the sharing point. Finally, off-chip main memory appears below the cache hierarchy.

The directory cache is accessed on data cache misses. To illustrate, Figure 1 shows three types of cache transactions, labeled "T1"–"T3." First, a transaction may miss all the way to main memory (T1), causing a new data block to be brought on chip. T1 transactions access the directory, but do not find the requested address tag–*i.e.*, they are directory cache misses. Second, a transaction may miss to the sharing point, but find its data on chip in a remote private cache (T2). T2 transactions are "sharing-based" transactions that require directory lookups to determine the kind of remote actions needed and the sharers involved. They are directory cache hits. Third, a transaction

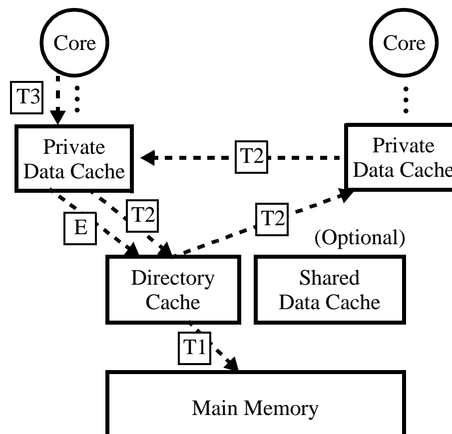


Fig. 1: Cache transactions in a multicore CPU: T1s fill entries into the directory cache, T2s and E's re-access existing directory entries, and T3s do not access the directory.

may hit in a core's private data cache (T3). T3 transactions do not access the directory. Besides data cache misses, the directory cache is also accessed on evictions that notify the directory. These are labeled "E" in Figure 1.

In addition to accessing the directory, data cache misses and evictions also change the directory's contents. T1 transactions allocate directory entries in the directory cache, initiating new *directory entry lifetimes*. Each directory entry starts out with a single sharer, but during its lifetime in the directory cache, its sharer list can be modified by T2 transactions. A T2 for a read request may add a sharer to the entry's sharer list, whereas a T2 for a write request sets the entry's sharer list to a single sharer (assuming invalidation on writes). Eviction notifications also change the entry's sharing degree, subtracting a sharer from the entry's sharer list. Finally, a directory entry's lifetime ends after all copies of its associated cache blocks have been evicted from the private data caches, potentially allowing the directory entry to be deallocated.

Notice, the T1–T3 and E accesses in Figure 1, as well as the changes they make to the directory's contents, are determined by the private data caches. Hence, they are architecture dependent. Specifically, scaling the number of private caches (*i.e.*, cores) and their capacities will affect the volume and distribution of T1, T2, T3, and E accesses. In turn, this will change the directory entry lifetimes within the directory cache as well as the entries' sharer lists. The goal of our work is to provide techniques for analyzing the directory's accesses and contents, especially as the private data cache hierarchy scales.

While understanding the scaling of both directory accesses and contents is important, the latter is particularly crucial. As discussed in Section 1, numerous techniques have tried to improve the scalability of directory caches. The effectiveness of these techniques hinges on the directory's contents–in particular, the number of live directory entries, the balance of private *vs.* shared entries, and among the shared entries, the degree of sharing. Our analyses will provide significant insights into these critical characteristics.

Finally, for some cache coherence protocols, our analyses are imprecise. In particular, there are protocols that do not notify the directory after certain data cache evictions–*e.g.*, eviction of shared (and clean) cache blocks. In this case, the directory cache may

retain entries whose lifetimes have ended, increasing the number of allocated entries which we do not analyze. Section 3.3 will discuss the impact of this on our analyses.

3. ANALYSIS FRAMEWORK

This section presents our RD-based framework for analyzing multicore scaling's impact on directory caches. Section 3.1 reviews multicore RD techniques. Then, Sections 3.2 and 3.3 develop new analyses to identify the directory's accesses and contents discussed in Section 2.

3.1. Multicore RD Analysis

Reuse distance has been used to analyze uniprocessor locality. For a sequential program, a reuse distance (RD) profile is a histogram of RD values for all memory references where each RD value is the number of unique data blocks referenced since the last reference to the same data block. (Reuse distance has also been called "stack distance" since RD calculations are performed on memory reference stacks [Mattson et al. 1970]). For a fully associative cache of size CS with an LRU eviction policy, references with RD $< CS$ are cache hits; hence, the cache miss count is the sum of all references in an RD profile above the RD value corresponding to capacity CS.

Although RD profiles model fully associative caches, most caches in real CPUs are set associative. Fortunately, RD profiles can accurately predict cache misses in set associative caches as well. This is because capacity misses usually dominate conflict misses. However, some errors will occur when applying RD profiling to set associative caches, which is an issue that we will address in Section 6. But while their predictions are not perfect, RD profiles are nevertheless attractive because they are *architecture independent*. So, a single profile can predict the misses for *any* cache size CS.

More recently, RD profiling has been extended for multicore processors by using parallel LRU stacks. For example, *private-stack reuse distance* (PRD) profiling [Schuff et al. 2009; Schuff et al. 2010; Wu and Yeung 2013; Wu et al. 2013] replicates LRU stacks, one per core, and plays each core's memory references on its local stack while maintaining coherence between all of the stacks. This technique can predict the cache misses occurring within private data caches.

To illustrate, Figure 2 shows a time-interleaved memory reference stream from two cores, C_1 and C_2. In the figure, memory blocks A, B, C, E, and F are referenced by core C_1 alone, while blocks G–K are referenced by core C_2 alone. Memory block D is *shared* and referenced by both cores. We assume the reference to block D at time $t = 13$ is a write (indicated by the superscript "W") whereas all other references to block D are reads unless otherwise specified. (The non-shared memory references can be either reads or writes without affecting the example). Given the reference stream in Figure 2, Figure 3 shows the corresponding parallel LRU stacks at different times.

In the absence of sharing, PRD profiling simply captures the intra-thread reuse that occurs within individual cores' memory reference streams, much like in a uniprocessor. For example, Figure 3(a) shows the state of C_1's LRU stack right before the reuse of A at $t = 8$. Since block A is found below blocks B–F in C_1's LRU stack, we say C_1's memory reference has PRD = 5. A cache of size 6 or more blocks would capture this reuse; otherwise, a cache miss would occur from C_1's private cache. Like sequential RD analysis, the histogram of all PRD values can predict a thread's private cache misses for *any* cache size.

But besides intra-thread reuse, PRD profiling also captures inter-thread interactions caused by sharing. In particular, for read sharing, PRD captures the resulting replication effects across LRU stacks. As Figure 2 shows, both C_1 and C_2 have read the shared block D by $t = 11$. Figure 3(b) illustrates the state of the corresponding LRU

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Core C_1:	A	B	C		D	E	F	A							B	D			A	C		E	D
Core C_2:				G					D	H	D	I	D^W	J			K	D			H		

Fig. 2: Two interleaved memory reference streams.

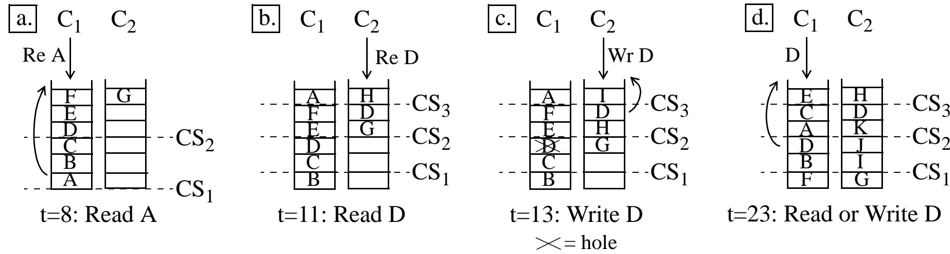


Fig. 3: LRU stacks illustrating (a) intra-thread reuse, (b) replication, (c) invalidation, and (d) another sharing pattern.

stacks. Notice, block D is replicated in the cores' stacks. PRD profiling accounts for the increased capacity pressure that such shared replicas can cause in the affected stacks.

PRD profiling also captures write sharing effects by maintaining coherence between LRU stacks. For example, consider C_2's write to block D at $t = 13$. In this case, invalidation occurs in C_1's stack, as shown in Figure 3(c). To prevent promotion of blocks further down the LRU stack, the invalidated block leaves behind a "hole" [Schuff et al. 2009]. Holes are unaffected by references to blocks above the hole, but a reference to a block below the hole moves the hole to where the referenced block was found. In our example, when C_1 references block B at $t = 15$, blocks A, F, and E in Figure 3(c) will be pushed down and the hole will move to depth 5 (B's old position), preserving the stack depth of block C. After the invalidation, C_1's re-reference of block D at $t = 16$ will miss regardless of the cache capacity–*i.e.* a coherence miss–so we say its PRD = ∞.

3.1.1. Core Count Scaling. The discussion above explains how PRD profiling can capture reuse and sharing for a fixed number of cores. In addition, PRD profiling can also analyze the impact of scaling core count. As the number of cores increases, PRD profiles often change systematically: they shift to larger RD values in a shape-preserving fashion [Wu and Yeung 2013; Wu et al. 2013]. At small cache sizes, profile shift is linear in the amount of core count scaling. But as cache size increases, the shift reduces, and becomes smaller for large caches. This is because memory references with small RD values tend to access private data, whereas shared references tend to exhibit larger RD values. Thus, core count scaling increases cache pressure more at small cache capacities. (Details on these effects can be found in prior work [Wu and Yeung 2013]).

Our research extends multicore RD analysis to handle directory caches. We develop several techniques for analyzing directory caches when scaling private cache capacity. We do not develop new techniques for analyzing core count scaling. Instead, in Section 5, we will use existing insights on core count scaling of PRD profiles, along with acquiring profiles at different numbers of cores, to reveal core count scaling's impact on directory caches.

3.2. Directory Access Analysis

As discussed in Section 2, directory accesses occur due to private data cache misses. Because PRD profiling can predict private data cache misses, it can identify directory accesses. In particular, given a reference's PRD and access mode (read or write), we can predict whether a cache miss and subsequent directory access will occur for a given cache size, CS. And if so, we can also predict its access type–T1, T2, or T3.

Consider the examples from Section 3.1. In Figure 3(a), if C_1's private cache is sufficiently large to capture the reuse on block A, then the reference hits–a T3 transaction–and no directory access occurs; otherwise, the reference misses and generates a directory access. For example, a cache size equal to CS_1 indicated in Figure 3(a) would result in a hit whereas a cache size equal to CS_2 would result in a miss. For the latter case, given there are no other copies of block A in Figure 3(a), this would be a T1 transaction that initiates a new directory entry lifetime.

In addition to capacity-induced T1 transactions, PRD profiling can also identify directory accesses due to sharing. For example, in Figure 3(c), if C_2's write to block D invalidates C_1's copy, this would generate a directory access to perform the inter-thread communication. In this case, the directory access would be a T2 transaction, reusing the directory entry for block D inserted at $t = 5$. Similarly, inter-thread communication from sharing may also occur at $t = 9$ (to forward block D from C_1 to C_2) and $t = 16$ (to forward block D from C_2 back to C_1 after the invalidation). These would also generate T2 transactions, though for reads rather than a write.

A critical observation, however, is whether or not these sharing-based communications occur is cache size dependent. Granted, sharing is an application-level property. But even if threads share data, the sharing may not manifest itself on chip if the cache is too small to capture the sharing pattern. So, the number of T2 transactions is in fact tied to temporal locality–in particular, to the *relative reuse distance between sharers*.

Notice, before the write in Figure 3(c), both cores have block D in their LRU stacks, but C_2 has referenced it more recently than C_1. So, the block is at different depths in the two stacks. Because there is a non-zero relative stack distance between the two copies, the behavior will depend on the private cache size. Figure 3(c) shows three cases, labeled CS_1–CS_3. If the cache size is CS_1, then both copies are on chip, thus capturing the sharing. In this case, C_2's reference generates the invalidation and a T2 transaction, as described above. But, if the cache size is CS_2, then only C_2's copy is on chip. We say block D is "temporarily private" [Alisafaee 2012]–*i.e.* it is private within the limited time window captured by CS_2. So, C_2's reference is a hit–a T3 transaction–with no directory access.

Figure 3(b) shows a similar situation. Like Figure 3(c), both cores in Figure 3(b) have block D in their LRU stacks at different stack depths. Again, if the cache size is CS_1, both copies are on chip and the sharing is manifested. But if the cache size is CS_2, only C_2's copy is on chip, so block D is temporarily private. One difference in Figure 3(b) compared to Figure 3(c), however, is the memory reference is a read rather than a write. So, even when sharing is captured on chip, the read still hits in the local private cache, a T3 transaction, and does not generate a T2 transaction.

To enable locality-aware sharing analysis, we introduce the notion of *remote reuse distance*, or PRD_{remote}. A memory reference's PRD_{remote} is the minimum stack depth across all remote LRU stacks. If $\text{PRD}_{remote} = \infty$, then the associated data block only resides in the core's local stack, and the memory reference is "truly private." If PRD_{remote} is finite, then its value specifies the capacity at which sharing is captured on chip. Given a private cache of size CS, $\text{PRD}_{remote} < CS$ would mean the sharing is captured; otherwise ($\text{PRD}_{remote} \geq CS$), the memory reference is temporarily private.

	Mode	PRD	PRD_{remote}	Example	Comment
T1 Transactions: New Lifetimes					
1	R	∞	∞	Similar to transaction #5	Cold Miss
2	W	∞	∞	Similar to transaction #6	Cold Miss
3	R	∞	$\geq CS$	Similar to transaction #7	Cold Miss
4	W	∞	$\geq CS$	Similar to transaction #8	Cold Miss
5	R	$\geq CS$	∞	Figure 3(a), CS_2	Truly Private
6	W	$\geq CS$	∞	Figure 3(a), CS_2, write case	Truly Private
7	R	$\geq CS$	$\geq CS$	Figure 3(b), CS_3	Temporally Private
8	W	$\geq CS$	$\geq CS$	Figure 3(c), CS_3	Temporally Private
T2 Transactions: Directory Reuses					
9	R	∞	$< CS$	Similar to transaction #10	Forwarding
10	R	$\geq CS$	$< CS$	Figure 3(d), CS_2	Forwarding
11	W	∞	$< CS$	Similar to transaction #12	Invalidation
12	W	$\geq CS$	$< CS$	Figure 3(d), CS_2, write case	Invalidation
13	W	$< CS$	$< CS$	Figure 3(c), CS_1	Invalidation
T3 Transactions: Data Cache Hits					
14	R	$< CS$	∞	Figure 3(a), CS_1	Truly Private
15	W	$< CS$	∞	Figure 3(a), CS_1, write case	Truly Private
16	R	$< CS$	$\geq CS$	Figure 3(b), CS_2	Temporally Private
17	W	$< CS$	$\geq CS$	Figure 3(c), CS_2	Temporally Private
18	R	$< CS$	$< CS$	Figure 3(b), CS_1	Read to Shared

Table I: The 18 possible data cache transactions enumerated by permuting the access mode (read or write) and the PRD and PRD_{remote} values ($< CS$, $\geq CS$, or ∞).

Finally, if the cache size in Figures 3(b) and (c) is CS_3, then not only is the sharing not manifested on chip, but *neither* copy of block D is on chip. In this case, C_2's reference (read or write) misses and generates a T1 transaction, much like the case for block A in Figure 3(a) assuming a CS_2 capacity.

3.2.1. Access Mode, PRD, PRD_{remote} Characterization. A memory reference's access mode, PRD, and PRD_{remote} values completely characterize its locality behavior within a private data cache hierarchy. As such, all possible data cache transactions, along with their directory accesses, can be enumerated by permuting the access mode (read or write) against the different PRD/PRD_{remote} outcomes discussed above ($< CS$, $\geq CS$, or ∞). Table I lists all of these permutations, giving rise to 18 distinct data cache transactions. The 18 cache transactions are grouped in terms of their directory access type: transactions 1–8 are the T1 transactions, transactions 9–13 are the T2 transactions, and transactions 14–18 are the T3 transactions (*i.e.*, no directory access).

All of the transactions in Table I either correspond to one of the examples in Figure 3, or are very similar to one of these examples. Specifically, transactions #5 and #14 correspond directly to Figure 3(a) assuming the cache sizes CS_2 and CS_1, respectively. Next, transactions #7, #16, and #18 correspond directly to Figure 3(b) assuming the cache sizes CS_3, CS_2, and CS_1, respectively. And, transactions #8, #13, and #17 correspond directly to Figure 3(c) assuming the cache sizes CS_3, CS_2, and CS_1, respectively. Lastly, transaction #10 corresponds to Figure 3(d) assuming the cache size CS_2.

A few of the transactions in Table I are similar to those mentioned above, except instead of performing reads, they perform writes. In particular, transaction #6 is similar to transaction #5, transaction #12 is similar to transaction #10, and transaction #15 is similar to transaction #14 in this fashion. And finally, several of the transactions in Table I are similar to those mentioned above, except instead of missing in the local cache because the accessed block resides below the cache size of interest (*i.e.*, PRD $\geq$ CS), the miss occurs because the block does not exist in the local cache at all (*i.e.*, PRD

= ∞). In particular, transactions #1, #2, #3, and #4 are similar to transactions #5, #6, #7, and #8, respectively, transaction #9 is similar to transaction #10, and transaction #11 is similar to transaction #12 in this fashion.

3.2.2. Evictions. In addition to T1 and T2 transactions, the directory is accessed on evictions as well, which PRD profiling can also predict. In particular, each memory reference pushes certain blocks in the local LRU stack downward. Whenever a block moves below a given stack depth, it is evicted from the cache with the corresponding capacity. For example, in Figure 3(d), the access to block D will push blocks E, C, and A down the stack. This will cause block E to be evicted from a cache of size CS_3, and block A to be evicted from a cache of size CS_2. As mentioned in Section 2, whether or not a particular eviction notifies the directory depends on the coherence protocol. The next section will address this issue.

3.3. Directory Contents Analysis

As discussed in Section 2, analysis of the directory's contents across CPU scaling is crucial for assessing the effectiveness of different directory techniques. Given the techniques presented in Section 3.2, we can already perform such analyses. In particular, each of the directory accesses in Table I (*i.e.*, the T1s and T2s) also modify the directory in a pre-determined way. So, once we extract a directory access stream from the PRD stacks, we can also track the directory contents associated with that stream.

This is done in the following manner. Initially, the directory cache is cleared of all directory entries. Each T1 transaction (#1–#8 in Table I) inserts a new entry with a single sharer, increasing the number of directory entries in the directory cache by one. Each T2 transaction reuses an existing directory entry, with reads (transactions #9 and #10) increasing the sharer count by one and writes (transactions #11, #12, and #13) setting the sharer count to one. Each data cache eviction that notifies the directory also reuses an existing directory entry, but decreases the sharer count by one. Lastly, if an entry's sharer count reaches zero, the number of directory entries in the directory cache decreases by one.

If all data cache evictions in the cache coherence protocol notify the directory (a common assumption made in many recent directory techniques [Alisafaee 2012; Ferdman et al. 2011; Sanchez and Kozyrakis 2012; Zebchuk et al. 2013]), our analysis exactly tracks the directory's contents. However, if some data cache evictions are silent, then our analysis is imprecise. For example, some cache coherence protocols do not inform the directory when evicting data blocks in shared state. In these cases, we can still identify the notifications, as well as the silent evictions. But the latter create dead directory entries that linger in the directory cache. Since we do not analyze directory cache eviction policies (this cannot be done in an architecture-independent fashion), we cannot determine when dead entries actually leave the directory cache.

4. DIRECTORY CACHE PROFILER

We implemented RD-based directory cache profiling within the Intel PIN tool [Luk et al. 2005]. We modified PIN to maintain coherent private LRU stacks and perform PRD profiling, as discussed in Section 3.1. (Our PIN-based profiler assumes 64-byte blocks in all of the LRU stacks). For every memory reference, our profiler consults the LRU stacks to compute the reference's PRD and PRD_{remote} values, and then uses Table I to determine the data cache transaction and directory access type. Figure 4 illustrates the structure of the profiler, showing the LRU stacks that it maintains.

To enable capacity scaling analysis, our PIN profiler refers to Table I multiple times per memory reference, determining the behavior for different CS values. While our framework allows exploring all CS exhaustively, we step CS in increments of 16KB

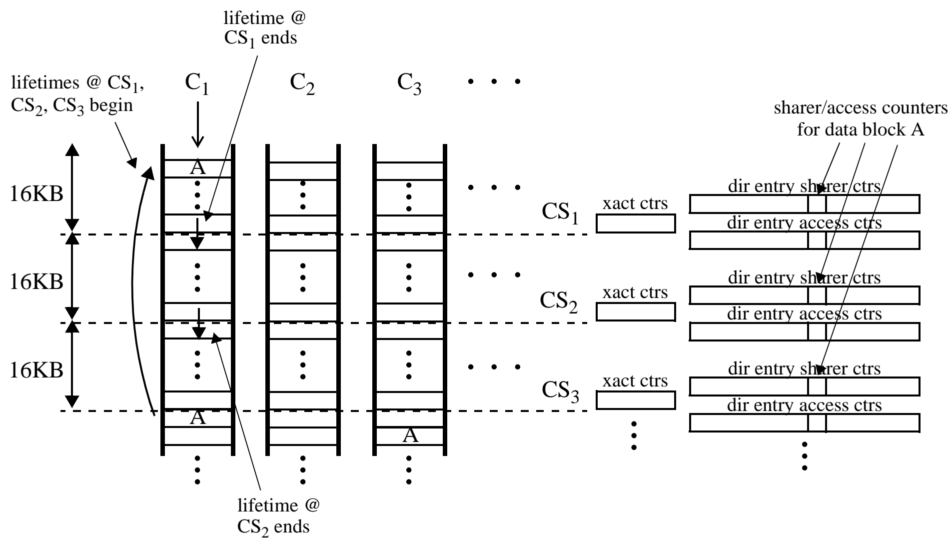


Fig. 4: LRU stacks and counters implemented in the PIN profiler.

and stop at the application's maximum PRD. For each CS value, we maintain 19 counters, one for each transaction in Table I plus one for evictions. After identifying the data cache transaction type at a particular CS value, the profiler increments the corresponding counter. Figure 4 illustrates the per-transaction counters, labeled "xact ctrs," one set for each profiled private cache size, labeled "CS_i" (CS_1, CS_2, CS_3, *etc.*).

In addition to counting transactions, our PIN profiler also tracks the directory cache contents. We maintain a set of sharer counters, *one per unique data block contained in all of the LRU stacks at every* CS_i. Figure 4 illustrates these counters, labeled "dir entry sharer ctrs." After updating the "xact ctr" at a particular CS_i, we check if the transaction causes a directory access, and if so, whether it changes the number of sharers. If the sharing degree changes, we modify the corresponding sharer counter. On each eviction, we also decrement the corresponding sharer counter for the evicted cache block. These sharer counters model a directory in which each unique data cache block allocates exactly one entry in the directory cache–for example, full-map directories or basic limited pointer directories. (Our profiler can also track the contents of more advanced directory techniques, which we will discuss in Section 7).

Our PIN profiler also counts accesses to individual directory entries during their lifetimes in the directory cache. We maintain another set of per-entry counters at every CS_i, labeled "dir entry access ctrs" in Figure 4. Each time a directory entry is accessed at a particular CS_i due to a T1 or T2 transaction, we increment the corresponding access counter. Also, each time an eviction occurs at a particular CS_i, we increment the corresponding access counter for the evicted block as well. Although many data cache transactions perform read-modify-write operations on the accessed directory entry, our profiler only attributes a single directory entry access for such transactions. (Section 7 will break down the reads and writes separately).

Sharer and access counters are allocated as memory references promote data blocks in the LRU stacks, initiating directory entry lifetimes at different cache sizes. In contrast, whenever a sharer counter decrements to zero, the corresponding directory entry's lifetime ends at the CS_i to which the sharer counter belongs. And, the corresponding access counter reflects the number of accesses the directory entry received

Benchmark	Suite	Problem Size	Footprint	Inst
fft (kernel)	SPLASH2	2^{22} elements	194.2	2.42
lu (kernel)	SPLASH2	2048^2 elements	32.3	22.95
radix (kernel)	SPLASH2	2^{24} keys	259.7	3.80
barnes	SPLASH2	2^{19} particles	106.8	32.67
fmm	SPLASH2	2^{19} particles	159.8	16.34
ocean	SPLASH2	1026^2 grid	228.9	1.40
water	SPLASH2	40^3 molecules	45.1	2.31
kmeans	MineBench	2^{22} objects, 18 features	304.1	10.16
blackscholes	PARSEC	2^{22} options	112.0	2.44
bodytrack	PARSEC	B_261,16k particles	3.5	10.27
canneal	PARSEC	2500000.net	98.4	0.10
fluidanimate	PARSEC	in_500k.fluid	92.6	3.33
raytrace	PARSEC	1920x1080 pixels	128.1	4.22
swaptions	PARSEC	2^{18} swaptions	666.7	22.41
streamcluster	PARSEC	2^{18} data points	41.6	4.33

Table II: Parallel benchmarks and the benchmark suites they belong to, the problem sizes and their memory footprints (in MB), and their resulting instruction counts (in billions).

during its lifetime for a private cache of size CS_i. We record this access count in a histogram for CS_i, and deallocate it (and its sharer counter) to reflect the directory entry's removal from the directory cache. Figure 4 illustrates such directory entry lifetime initiation and termination. In the figure, block A starts out at a stack depth below CS_3 in core C_1 and C_2's LRU stacks. (We assume block A resides in only two LRU stacks). At some point, core C_1 references block A, initiating directory entry lifetimes at capacities CS_1, CS_2, and CS_3, and allocating sharer and access counters at those three capacities. Then, Figure 4 shows the first lifetime terminates when the block is pushed below capacity CS_1, and the second lifetime terminates when the block is pushed below capacity CS_2, causing deallocation of sharer and access counters when the block moves past each capacity.

Finally, our PIN profiler follows McCurdy's method [McCurdy and Fischer 2005] 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 homogeneous threads, this approach yields profiles that accurately reflect locality on real CPUs [Jiang et al. 2010; Wu and Yeung 2013], especially for PRD profiles. Also, our profiler dumps its counter values at the end of each program run. Hence, our current profiles only reflect entire program behavior. They do not capture the effect of different program phases.

5. PROFILE STUDIES

With our profiler, we study the impact of multicore scaling on directory caches using 15 parallel benchmarks. Table II lists the benchmarks and the benchmark suites that they belong to (either SPLASH2 [Woo et al. 1995], MineBench [Narayanan et al. 2006], or PARSEC [Bienia et al. 2008]). The middle column in Table II specifies the problem sizes used for each benchmark, while the second-to-last column reports the total memory footprint size (in MB). Notice, we only consider a single problem size per benchmark, so our experiments do not consider the impact of problem scaling. Finally, the last column in Table II reports the resulting instructions executed (in billions) given the specified problem sizes. For the kernels–fft, lu, and radix–we profiled the entire

benchmark. For the other benchmarks, we ran the first parallel iteration to warm up the PRD stacks, and then profiled the second parallel iteration.

Our study proceeds in three parts. First, we study how scaling affects the directory's access stream (Section 5.1). Then, we study how scaling affects the directory's contents (Section 5.2). Lastly, we study the distribution of accesses across the directory to show temporal reuse of directory entries (Section 5.3). In all three parts, we first address cache size scaling, and then we examine core count scaling.

5.1. Study 1: Directory Access Count Results

Figure 5 shows the impact of scaling private data cache size on the number of directory cache accesses for all 15 benchmarks. In particular, the solid lines in Figure 5, labeled "Total Misses," report the total number of cache miss-induced directory accesses (*i.e.*, T1 + T2 tranactions) incurred by a 64-core CPU as private cache size is varied from 16 KB per core to the maximum PRD observed for each benchmark, usually between 2-4 MB per core. We normalize all directory access counts by the total number of instructions executed in each benchmark and multiply by 1000, so the Y-axes in Figure 5 report directory accesses per 1000 instructions, or "APKI." These results were obtained using the "xact ctrs" from our profiler.

In Figure 5, we can see that the number of directory cache accesses is highly sensitive to data cache size: for most benchmarks, *it drops rapidly as the data cache capacity increases*. The drop is particularly pronounced in lu, fmm, ocean, water, bodytrack, fluidanimate, and streamcluster. It is also very large in fft, radix, barnes, blackscholes, and raytrace, though for these benchmarks, most of the drop occurs around 16 KB. Table III provides a few of the actual numbers behind the trends that are visible in Figure 5. In particular, columns 2–4 of Table III report the cache miss-induced directory APKI at 16 KB, 256 KB, and 1 MB per core for every benchmark. At 16 KB, about half of the benchmarks exhibit a directory APKI exceeding 11 (reaching 32 in one case). But at 1 MB, all benchmarks except for canneal exhibit a directory APKI of only 7.1 or less, with half under 1 APKI. Across all of the benchmarks, the average directory cache APKI drops from 5.3 at 16 KB to 1.6 at 1 MB, a factor of 3.3x.

This drop is not surprising since one would expect a reduction in data cache misses to occur when scaling the private data caches. But the full story is actually more nuanced, reflecting on how cache size scaling affects on-chip sharing. To illustrate, two other curves in Figure 5 break down the cache miss-induced directory accesses into different types. In particular, the dashed lines in Figure 5, labeled "T2," plot the APKI for T2 transactions only. (Hence, the gap between the "Total Misses" and "T2" lines break down the APKI for T1 transactions). Also, the dot-dashed lines, labeled "T2 Read Shared," plot the APKI for T2 transactions associated with read sharing–*i.e.*, transaction #10 in Table I.[1]

At small cache sizes, Figure 5 shows the directory cache's accesses are dominated by T1 transactions–*i.e.*, the gap between the "Total Misses" and "T2" curves is large. For most benchmarks, there is a lack of T2 transactions near 16 KB, which makes sense since these data caches are too small to capture the shared accesses occurring between threads. So, the majority of references are to private data, regardless of whether they are truly or temporarily private. As data cache size increases, two trends occur. First, truly private data blocks begin fitting in cache, decreasing the number of T1 transactions. But second, temporarily private data blocks begin manifesting their sharing

[1]Note, transaction #9 in Table I also includes T2 transactions associated with read sharing (cold misses to read-shared blocks), but these are a relatively small component. Instead, transaction #9 is mostly associated with read-write sharing in which a read is performed to a previously invalidated block. So, we omit transaction #9 from the "T2 Read Shared" lines in Figure 5.

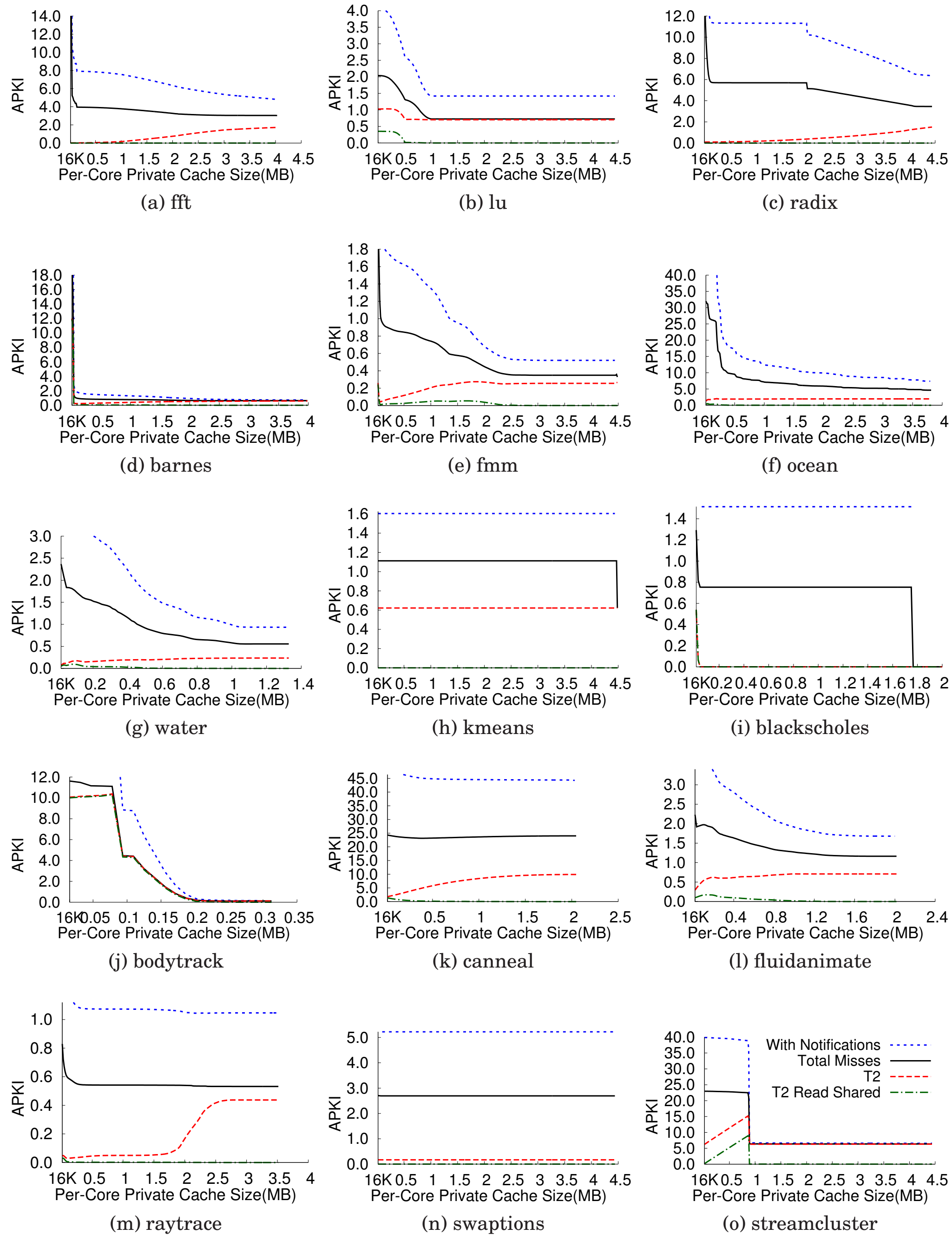


Fig. 5: Directory cache APKI *v.s.* private cache size for all accesses (including cache eviction notifications), cache miss-induced accesses, T2 transactions, and T2 transactions associated with read sharing. All results are for 64-core CPUs.

Benchmark	Cache Miss APKI			T2	APKI	
	16 KB	256 KB	1 MB	∞	16 cores	256 cores
fft	16.0	3.9	3.8	1.7	3.7	3.9
lu	2.0	1.9	0.7	0.7	0.2	1.1
radix	16.7	5.7	5.7	2.3	5.6	6.1
barnes	19.1	0.9	0.8	0.6	0.6	0.9
fmm	2.1	0.9	0.7	0.3	0.7	0.9
ocean	32.0	15.9	7.1	2.0	6.0	10.1
water	2.4	1.5	0.6	0.2	0.5	0.8
kmeans	1.1	1.1	1.1	0.6	1.1	0.6
blackscholes	1.3	0.8	0.8	0.0	0.8	0.8
bodytrack	11.6	0.1	0.1	0.1	0.1	0.3
canneal	24.3	23.3	23.6	9.9	22.9	24.9
fluidanimate	2.2	1.8	1.3	0.7	0.8	1.9
raytrace	0.8	0.6	0.5	0.4	0.5	0.5
swaptions	2.7	2.7	2.7	0.2	2.6	2.9
streamcluster	23.0	22.9	6.4	6.3	5.7	6.9
Average	5.3	2.2	1.6	0.5	1.3	1.8

Table III: Cache miss-induced directory APKI for three private cache sizes and APKI for intrinsic coherence-related directory accesses (both for 64 cores), and APKI for 16- and 256-core CPUs with 64 MB of total private cache.

patterns on chip, increasing the number of T2 transactions. This causes the steady rise in the “T2” curves that is visible for many of the benchmarks in Figure 5.

While T2 transactions generally go up with capacity scaling, they can also drop due to read sharing. Figure 5 shows the “T2 Read Shared” transactions increase as more remote sharers are captured on chip. But once *all sharers* are cached, these directory accesses are eliminated–*i.e.* the read-sharing working set fits in cache. This phenomenon is clearly visible in lu, bodytrack, and streamcluster (and to a lesser extent in fmm, water, and fluidanimate). In contrast, write sharing leads to coherence-related T2 transactions. These also increase with capacity scaling, but they cannot be eliminated by capturing all of the sharers on chip. This is why many of the “T2” curves in Figure 5 rise continuously, as mentioned above.

Notice, at each benchmark’s maximum PRD, all read-shared T2s are eliminated while all write sharing is exposed. These “∞” private caches quantify a program’s intrinsic coherence-related directory accesses. The column labeled “T2-∞” in Table III reports these directory accesses. On average, they only reach 0.5 APKI. *Hence, while scaling the data caches exposes sharing-based misses to the directory, it also decreases misses to truly private data by a far larger amount*. This is why overall, data cache size scaling reduces the total number of directory accesses.

In addition to data cache size scaling, the directory access stream is also affected by core count scaling. To illustrate, Figure 6 shows the same cache-miss induced directory access curves from Figure 5–*i.e.*, the “Total Misses” curves–at three different core counts: 16, 64, and 256. These curves were obtained by performing separate profiling runs, one for each core count. (Note, the X-axes in Figure 6 plot *total* cache size, instead of *per-core* cache size as is done in Figure 5, to facilitate comparisons across core count). Only 6 benchmarks are shown since the behavior is very similar across all 15 benchmarks.

Figure 6 shows core count scaling shifts the directory access curves to larger RD values in a shape-preserving fashion. This is the same behavior that PRD profiles exhibit, as shown in previous research (see Section 3.1), which makes sense since directory accesses are derived from private cache misses that PRD profiles capture. Overall, the

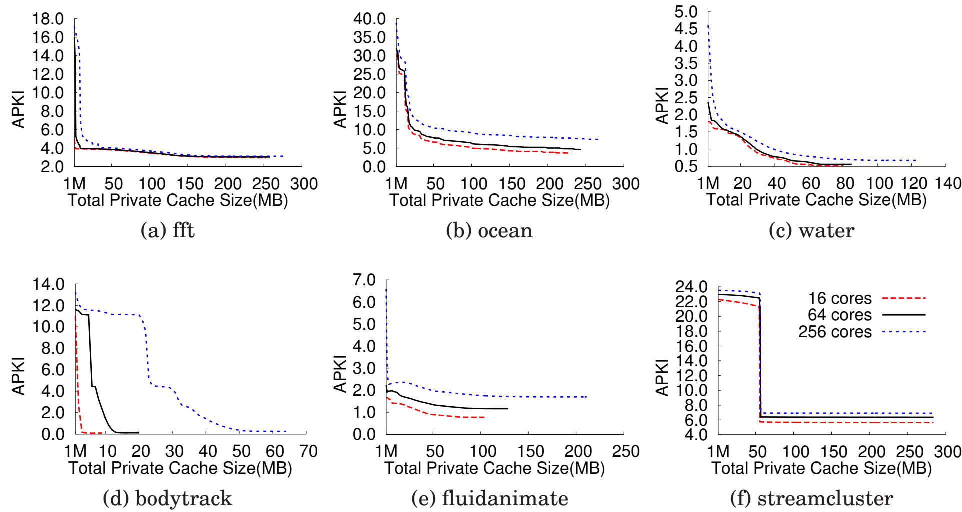


Fig. 6: Total cache miss-induced directory APKI for 16-, 64-, and 256-core CPUs.

shift increases the directory accesses at a given cache size, but in most cases the impact is small. For example, the last two columns of Table III report the directory APKI for 16- and 256-core CPUs that employ 64MB of total private cache. As Table III shows, the directory cache accesses only increase from 1.3 to 1.8 APKI on average–*i.e.*, by 38.5%–despite a 16x scaling in core count.

Finally, while we have focused on cache miss-induced directory accesses, the directory access stream also contains cache eviction notifications. In Figure 5, the dotted lines labeled "With Notifications" plot directory accesses when notifications are added. (Note, some of these curves are clipped at small cache sizes because we limited the maximum Y-axis value to make important behaviors at larger cache sizes visible). In many benchmarks, notifications roughly double the number of directory accesses at small cache sizes. This is because small data caches contain mostly private data blocks, each incurring a T1 transaction to insert its directory entry into the directory cache and a notification to end the entry's lifetime. The pairing of notifications with T1s causes the doubling. (In some benchmarks, the notifications more than double the number of directory accesses because they are for T1 transactions from the warmup region which are not counted in the region of execution that we profiled.) For larger caches, data blocks may incur many T2 transactions not paired with evictions; hence, notifications comprise a smaller fraction of the directory access stream as caches scale. But notifications do not change the main point: cache size scaling significantly reduces the number of directory cache accesses even when accounting for notifications.

5.2. Study 2: Directory Contents Results

Figure 7 shows the impact of scaling private data cache size on the number of directory entries in the directory cache for all 15 benchmarks. In particular, the solid lines, labeled "Total," plot the ratio of total live directory entries to total private cache blocks–a metric called *coverage* [Sanchez and Kozyrakis 2012]–as data cache size varies. (Coverage at each cache size is time-averaged across the entire profiling run). The number of live directory entries is obtained by tracking the number of sharer counters in our profiler. All results are for 64-core CPUs.

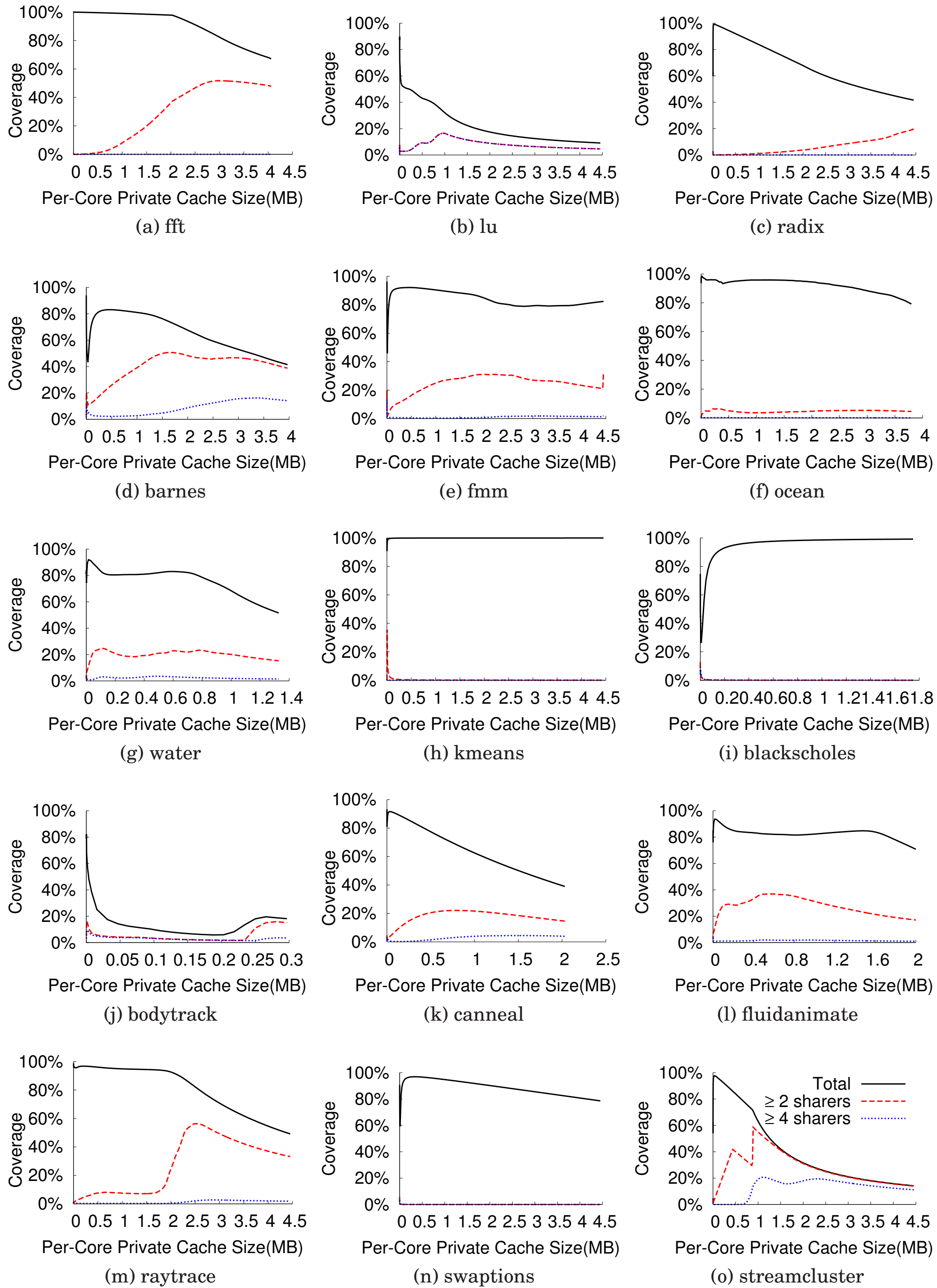


Fig. 7: Coverage breakdown *vs.* private data cache size based on sharers. Dashed lines break down coverage for single- *vs.* multi-sharer entries. Dotted lines break down coverage for entries with 4 or more sharers. All results are for 64-core CPUs.

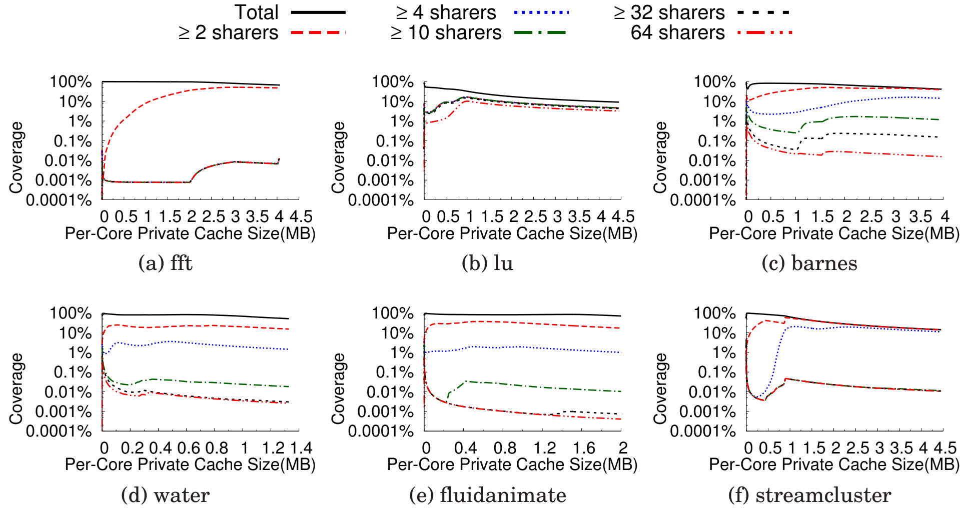


Fig. 8: Same as Figure 7(a), (b), (d), (g), (l), and (o) except additional lines further break down coverage for entries with $\geq$ 10, $\geq$ 32, and 64 sharers. Y-axes are on a log scale.

As Figure 7 shows, *coverage–and hence, the portion of on-chip memory needed to store the active directory entries–often decreases significantly with cache size scaling*. In most cases, coverage starts near 100%, but then drops noticeably as cache size increases. The magnitude of the drop is highly application dependent. For kmeans and blackscholes, there is no drop and coverage remains at about 100%. For extreme cases like lu, bodytrack, and streamcluster, coverage dips below 20% (a $>$ 80% drop). For about half the benchmarks, there is a large drop, though not as extreme as 80%. On average, the coverage at each benchmark's maximum PRD is 56.7%–*i.e.*, a little less than half (43.3%) of the private caches' data blocks are not associated with unique directory entries. As we will see in Section 7, such large drops in coverage have implications for the directory cache's capacity.

The drop in coverage observed in Figure 7 is primarily due to increased sharing that occurs at larger cache sizes. As discussed in Section 5.1, T2 accesses are negligible at small cache sizes, but go up with data cache size scaling because applications' sharing patterns become exposed on chip. These extra T2s tend to increase the sharers tracked per directory entry. *So, while single-sharer entries dominate at small cache sizes, multi-sharer entries become significant at large cache sizes*. Since shared data blocks can be tracked with fewer directory entries compared to private-only blocks, this causes the directory's coverage to go down.

To illustrate, the dashed lines in Figure 7 labeled "$\geq$ 2 sharers" plot coverage for the directory entries with 2 or more sharers, as tracked by the sharer counters in our profiler. (So, the gap between the "Total" and "$\geq$ 2 sharers" curves breaks down the coverage for single-sharer entries). For 64 KB private caches, only 9.3% of directory entries are multi-sharer entries (over 90% are private entries) averaged across all benchmarks. But by 1 MB, 28.0% are multi-sharer entries on average, and at each benchmark's maximum PRD, 39.1% are multi-sharer entries. As we will see in Section 7, this increase in multi-sharer entries (and decrease in single-sharer entries) can have a significant impact on the size of certain directory cache techniques.

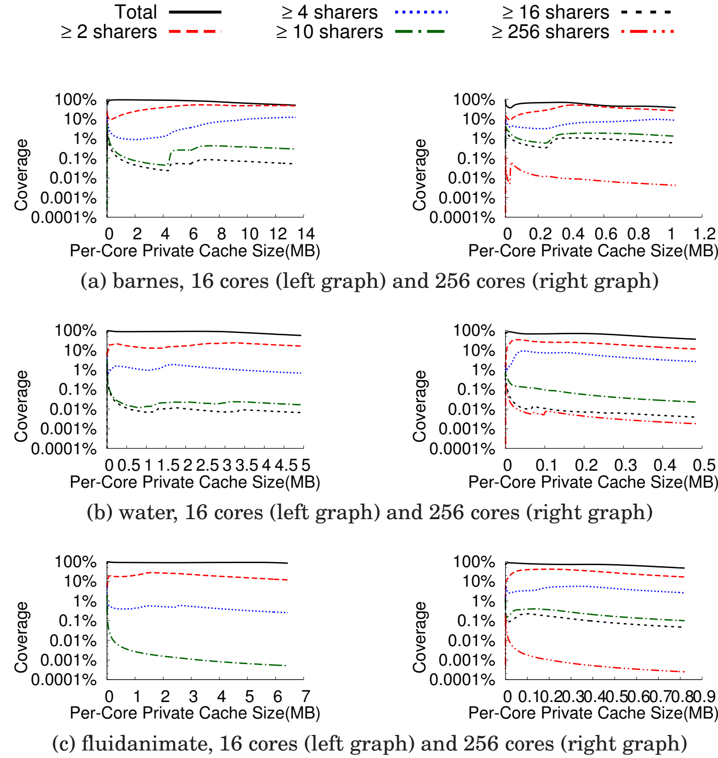
Interestingly, the increase in sharing occurs non-uniformly. Figure 7 illustrates this by plotting the coverage for directory entries with 4 or more sharers (the dotted lines

Benchmark	≥32 Sharers	Coverage Drop 256 KB	1 MB
fft	0.0008%	0.2%	0.2%
lu	14.9%	1.1%	-8.4%
radix	0.001%	0.6%	0.2%
barnes	0.03%	28.7%	13.8%
fmm	0.01%	11.7%	7.0%
ocean	0.003%	4.6%	4.1%
water	0.004%	14.1%	3.9%
kmeans	0.0003%	0.3%	0.1%
blackscholes	0.02%	16.9%	4.2%
bodytrack	1.2%	8.9%	10.5%
canneal	0.02%	1.9%	0.4%
fluidanimate	0.0008%	2.3%	10.3%
raytrace	0.006%	1.7%	8.2%
swaptions	0.009%	6.2%	1.3%
streamcluster	0.04%	0.6%	-0.3%
Average	0.01%	2.7%	1.8%

Table IV: Coverage of directory entries with 32 or more sharers for 64-core CPUs with 1 MB of private cache, and drop in coverage due to scaling from 64 to 256 cores for 256 KB and 1 MB private caches.

labeled "$\geq$ 4 sharers"). As Figure 7 shows, 2- and 3-sharer entries (*i.e.*, the gap between the "≥ 2 sharers" and "≥ 4 sharers" curves) account for the majority of multi-sharer entries created by cache size scaling. In contrast, directory entries with many sharers are negligible. To illustrate further, Figure 8 plots additional lines for six of the benchmarks, breaking down the directory entries with 10 or more sharers, 32 or more sharers, and 64 sharers. (Note, the Y-axes are now on a log scale). As Figure 8 shows, directory entries with wide sharing occupy a very small portion of the directory. For example, except for lu, the directory entries with 32 or more sharers in Figure 8 account for less (and in most cases, far less) than 1% coverage, and do not increase with cache size scaling. And although not shown in Figure 8, we find the same behavior occurs in the other 9 benchmarks as well. Table IV provides a few actual numbers. In particular, the second column of Table IV reports the coverage for $\geq$ 32 entries at a private cache size of 1 MB across all benchmarks. For 13 out of 15 benchmarks, the coverage is miniscule. (For lu and bodytrack, the coverage is 14.9% and 1.2%, respectively). On average, Table IV shows the $\geq$ 32 entries account for only 0.01% coverage at 1 MB. *So, the reduction in coverage due to cache size scaling primarily comes from increasing directory entries with a few sharers, not from increasing widely shared entries*. This suggests that the increase in sharing is mainly due to write sharing which tends to limit the sharing degree. There is definitely only small amounts of wide read sharing in our benchmarks. One useful byproduct of this result is that there will likely be a one-to-one correspondence between unique block addresses and directory entries. (Most directory entry formats track small degrees of sharing using a single directory entry). So, while the coverage results in Figure 7 are for full-map directories, they also reflect coverage in other types of directories as well (*e.g.*, limited directories) due to the dominance of narrow sharing. We will see this effect in Section 7.2.

While increased sharing is the main reason for the drop in coverage at larger cache sizes, another reason is changing working sets. As mentioned earlier, we warm up our profiler's PRD stacks during the first parallel iteration, and then profile the second parallel iteration. In some cases, the second iteration may access data that is different from the first iteration, bringing many new directory entries into the directory cache



(a) barnes, 16 cores (left graph) and 256 cores (right graph)

(b) water, 16 cores (left graph) and 256 cores (right graph)

(c) fluidanimate, 16 cores (left graph) and 256 cores (right graph)

Fig. 9: Same as Figure 8(c), (d), and (e) except for 16 and 256 cores.

during the profile period. Because we compute coverage in a time-averaged fashion, the delay in populating the directory cache artificially reduces coverage. Moreover, this effect is more pronounced at larger data cache sizes where it takes longer to bring in all of the new directory entries. This is why coverage drops in swaptions (Figure 7n) despite the lack of sharing in that benchmark. But across the other benchmarks, changing working sets do not play a major role in affecting coverage.

In addition to data cache size scaling, the directory cache contents are also affected by core count scaling. Like cache size scaling, core count scaling also increases on-chip sharing, so it too lowers coverage and reduces the on-chip memory needed for the directory. Somewhat surprisingly, however, the impact from core count scaling is much less compared to cache size scaling. Columns 3 and 4 of Table IV report the percentage drop in coverage when scaling from 64 to 256 cores observed at private cache sizes of 256 KB and 1 MB, respectively. In most cases, coverage changes by only a few percent. For a small number of cases, the reduction exceeds 10%, but on average, there is only a 2.7% drop at 256 KB and a 1.8% drop at 1 MB. This is much less than the 43.3% drop in coverage shown in Figure 7 across cache size scaling.

Core count scaling's impact on coverage is limited because it tends to increase sharing more for data that are widely shared, which are in the minority. To illustrate, Figure 9 plots a similar coverage breakdown as Figure 8, except at different core counts. In particular, the left set of graphs in Figure 9 plots the breakdown for 16 cores while the right set of graphs plots the breakdown for 256 cores (instead of for 64 cores as is done in Figure 8). These core count sensitivity results are shown for three benchmarks–barnes, water, and fluidanimate–corresponding to Figures 8(c), (d), and

(e). As Figure 9 shows, the coverage for widely shared entries increases noticeably with core count scaling, but the entries with 2–3 sharers do not increase as much. Since the former is a small part of the directory cache, the overall impact of core count scaling on coverage is small.

5.3. Study 3: Access Distribution Results

Finally, we study the distribution of accesses over individual directory entries. We observe that the accesses a directory entry receives are related to its sharing degree. Entries with a few sharers tend to receive a few accesses whereas entries with wide sharing tend to receive many more accesses (during the entries' lifetimes in the directory cache). Recall from Section 5.2 that multicore scaling increases sharing non-uniformly. As sharing goes up, the number of widely shared entries grows slower than the number of entries with just a few sharers. *So, like sharing, reuse of directory entries also tends to be non-uniform, with a small part of the directory–i.e., the entries with wide sharing–receiving a disproportionately large fraction of the directory accesses.*

Figure 10 illustrates this point. In the figure, we break down the directory's coverage, just like in Figure 7. But instead of sharing degree, Figure 10's breakdown is across different numbers of cache miss-induced accesses that entries receive during their lifetimes in the directory cache. In particular, the solid lines in Figure 10, labeled "Total," plot the total coverage for the directory cache. (These curves are identical to the corresponding curves in Figure 7). Then, the dashed lines labeled "$\geq$ 2 accesses", dotted lines labeled "$\geq$ 4 accesses", and dot-dashed lines labeled "$\geq$ 10 accesses" break down the coverage for entries with 2 or more, 4 or more, and 10 or more cache miss-induced accesses, respectively. These results were obtained using the access counters from our profiler. All results are for 64-core CPUs.

Notice, the breakdown curves in Figure 10 are very similar to those in Figure 7. For instance, the breakdown for single-access entries (the gap between the "Total" and "$\geq$ 2 accesses" curves) in Figure 10 are identical to the breakdowns for single-sharer entries (the gap between the "Total" and "$\geq$ 2 sharers" curves) in Figure 7.[2] Also, the breakdowns for the multi-access entries in Figure 10 are quite similar to those for the multi-sharer entries in Figure 7. Some discrepancy occurs in barnes, fmm, fluidanimate, and streamcluster, but for the most part, the "$\geq$ 4 accesses" and "$\geq$ 10 accesses" curves in Figure 10 resemble the "$\geq$ 4 sharers" and "$\geq$ 10 sharers" curves in Figure 7. Hence, just like wide sharing is concentrated on a small number of directory entries, so too are many accesses concentrated on a small number of directory entries.

Table V shows this leads to significant reuse of the multi-access directory entries. In particular, columns 2–4 of Table V report the percentage of directory accesses that are destined to entries receiving $\geq$ 3 accesses during their lifetimes. Results are shown for 256 KB, 1 MB, and ∞ private caches. We also report in columns 5–7 the percentage of all directory entry lifetimes that the $\geq$ 3 lifetimes represent. At 256 KB, Table V shows the entries with $\geq$ 3 accesses account for only 5.4% of all directory entry lifetimes; yet, they receive 24.3% of all directory accesses (on average). At 1 MB, they account for 23.0% of lifetimes, but receive 42.7% of the accesses. And at ∞, they account for 35.0% of lifetimes, but receive 57.3% of the accesses.

While these results consider all cache miss-induced directory accesses (T1 and T2 transactions), we find the reuse of multi-access entries is even greater when considering just the T2 transactions. In the last two columns of Table V, we report the per-

[2]This is because directory entries for private data blocks receive exactly one access during their lifetime in the directory cache: a T1 transaction to fill the entry into the directory cache. So, there is a one-to-one correspondence between the number of private block directory entries and the number of accesses to such directory entries.

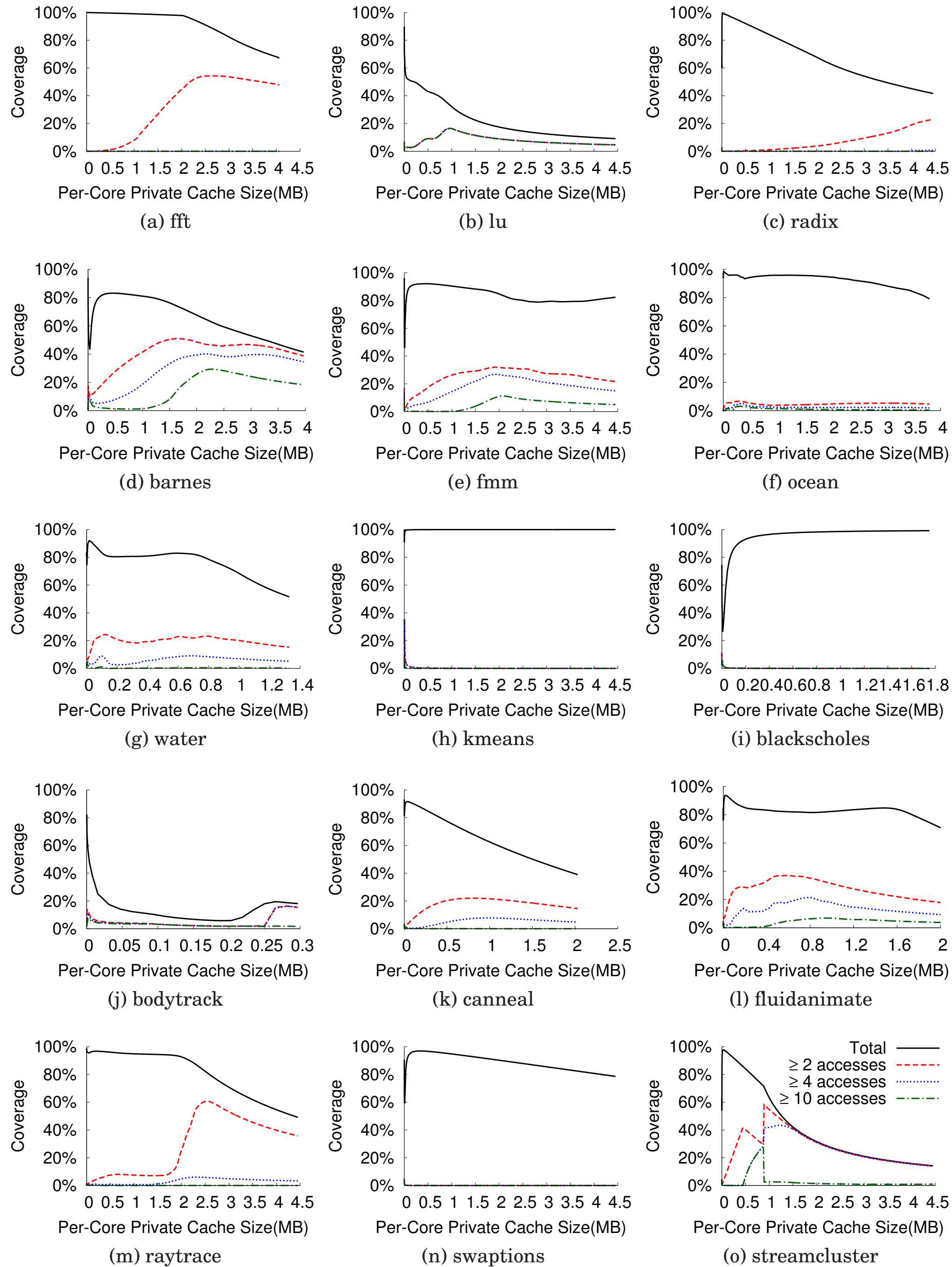


Fig. 10: Coverage breakdown *vs.* private data cache size based on accesses. Dashed, dotted, and dot-dashed lines break down coverage for entries that received 2 or more, 4 or more, and 10 or more accesses, respectively. All results are for 64-core CPUs.

Benchmark	% Accesses to ≥3 Entries			% Entries with ≥3 Accesses			% T2 Accesses	
	256 KB	1 MB	∞	256 KB	1 MB	∞	256 KB	1 MB
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%	100%
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%	31.0%	55.8%	5.2%	15.8%	18.1%	69.5%	79.1%
ocean	13.4%	27.9%	44.4%	5.2%	3.3%	4.9%	95.5%	97.5%
water	11.4%	43.5%	45.5%	6.3%	17.6%	17.6%	61.6%	83.8%
kmeans	70.7%	67.7%	56.7%	0.6%	0.1%	0.0%	100%	100%
blackscholes	0.8%	0.6%	0.4%	0.1%	0.0%	0.0%	99.6%	99.6%
bodytrack	80.8%	99.3%	99.3%	11.4%	85.2%	85.2%	99.7%	100%
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	6.2%	5.3%	15.4%	0.8%	0.6%	6.7%	68.4%	47.1%
swaptions	6.3%	6.3%	6.3%	0.0%	0.0%	0.0%	100.%	100%
streamcluster	26.9%	98.9%	99.9%	0.2%	73.9%	98.4%	70.0%	99.9%
Average	24.3%	42.7%	57.3%	5.4%	23.0%	35.0%	84.6%	83.1%

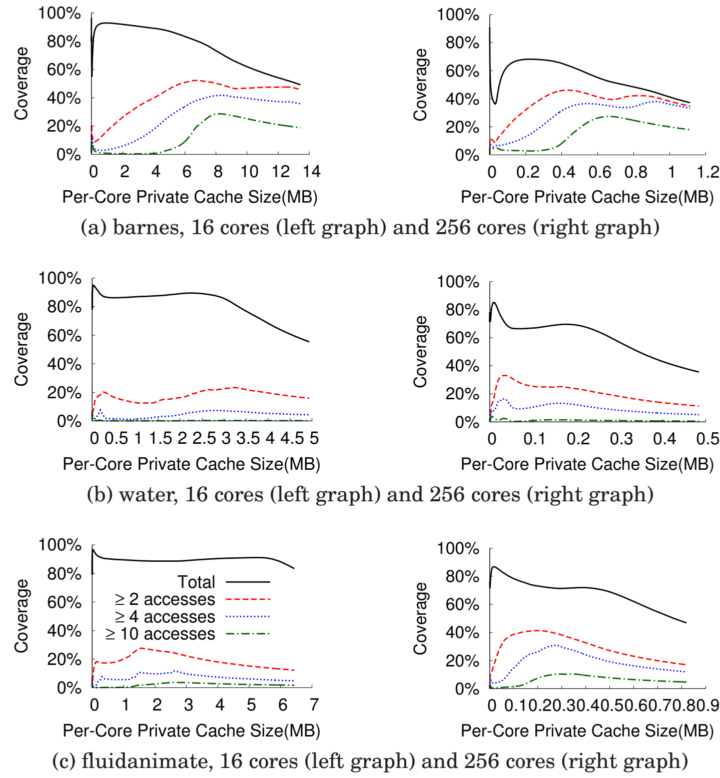
Table V: Percentage of directory accesses destined to $\geq$ 3-access entries, percentage of directory entries with $\geq$ 3 accesses, and percentage of T2 transactions destined to $\geq$ 3-access entries.

centage of T2 transactions that are destined to entries with $\geq$ 3 accesses for 256 KB and 1 MB private caches. These results show the great majority of T2s, 84.6% and 83.1%, are captured by a minority of directory entry lifetimes, 5.4% and 23.0%, respectively. As discussed in Section 2, T2s are on-chip transactions whereas T1s can be off-chip transactions. Hence, T2-induced directory accesses are more latency sensitive. (We will discuss this issue further in Section 7.4). Our results show *a small fraction of the directory cache can service the majority of latency-sensitive directory accesses*.

In addition to data cache size scaling, core count scaling also affects the distribution of accesses across the directory's entries. As discussed in Section 5.2, core count scaling increases on-chip sharing, especially for widely shared data blocks. This results in even more accesses to the widely shared directory entries, and an even more disproportionate reuse of those entries. Figure 11 illustrates this point for three representative benchmarks–barnes, water, and fluidanimate. In Figure 11, we plot the exact same coverage breakdown as in Figure 10, except we do it for 16 cores (left set of graphs) and 256 cores (right set of graphs) instead of for 64 cores as is done in Figure 10.

Figure 11 shows core count scaling reduces the overall coverage in the directory cache, as can be seen by comparing the "Total" curves at 16 *vs.* 256 cores. This is because core count scaling reduces the number of directory entries as a consequence of increasing sharing, an observation already made in Section 5.2. At the same time, Figure 11 also shows the number (and coverage) of directory entries receiving multiple accesses increases as well. The effect is particularly noticeable in water and fluidanimate. However, this increase is quite small, especially considering that it comes from a 16x scaling in core count. *So, while the temporal reuse of multi-sharer directory entries increases with core count scaling, the overall impact is not significant.*

For example, Table VI shows the impact of scaling core count on the results from Table V. In particular, columns 2 and 3 of Table VI report the difference in percentage of accesses destined to directory entries with 3 or more accesses at 16 and 256 cores, respectively, compared to 64 cores assuming 64 MB of private cache. (In other words, these columns report the change relative to the data in column 3 of Table V). And, columns 4 and 5 of Table VI report the difference in percentage of T2 transactions, also destined to directory entries with 3 or more accesses at 16 and 256 cores, respectively,



(a) barnes, 16 cores (left graph) and 256 cores (right graph)

(b) water, 16 cores (left graph) and 256 cores (right graph)

(c) fluidanimate, 16 cores (left graph) and 256 cores (right graph)

Fig. 11: Same as Figure 10(d), (g), and (l) except for 16 and 256 cores.

Benchmark	Difference in % of Accesses to ≥3 Entries		Difference in % of T2 Accesses to ≥3 Entries	
	16 cores	256 cores	16 cores	256 cores
fft	-0.3%	1.3%	-7.6%	24.6%
lu	-3.2%	-8.7%	0.0%	-0.6%
radix	-1.6%	2.9%	-43.7%	16.8%
barnes	-5.4%	2.7%	-1.6%	0.9%
fmm	-2.7%	4.3%	0.7%	0.8%
ocean	-14.3%	19.5%	-1.3%	0.6%
water-spatial	-5.7%	8.3%	-7.2%	8.3%
kmeans	-12.9%	-37.7%	0.0%	0.0%
blackscholes	-0.4%	1.7%	0.0%	0.0%
bodytrack	0.5%	-0.5%	0.0%	0.0%
canneal	-2.0%	0.6%	-1.3%	0.4%
fluidanimate	-23.6%	8.5%	-5.6%	-0.3%
raytrace	-0.2%	4.4%	0.5%	9.3%
swaptions	-2.2%	5.6%	0.0%	0.0%
streamcluster	0.4%	-0.1%	0.0%	0.0%
Average	-4.9%	0.8%	-4.5%	4.1%

Table VI: Difference in percentage of directory accesses destined to $\geq$ 3-access entries, and difference in percentage of T2 transactions destined to $\geq$ 3-access entries for 16 and 256 cores compared to 64 cores. All results assume 64 MB of private cache.

compared to 64 cores assuming 64 MB of private cache. (In other words, these columns report the change relative to the last column of data in Table V). As Table VI shows, core count scaling only changes the results from Table V by a few percent or less in most cases. On average, scaling up to 256 cores (from 64 cores) increases the reuse of $\geq$ 3-access entries from all accesses and from T2 transactions by only 0.8% and 4.1%, respectively, while scaling down to 16 cores decreases the reuse of $\geq$ 3-access entries from all accesses and from T2 transactions by only 4.9% and 4.5%, respectively.

6. VALIDATION

As mentioned in Section 3.1, RD analysis only approximates cache behavior because real hardware caches are set associative, but the LRU stacks employed in RD analysis are fully associative. In this section, we quantify this source of error by comparing the profile predictions from Section 5 against cache simulations. We first discuss our simulation methodology in Section 6.1. Then, Sections 6.2–6.4 present the validation experiments for each of the major results from Section 5. Next, Section 6.5 discusses the impact that timing potentially has on our validation results. Finally, while our main focus is on validating accuracy, Section 6.6 also quantifies the speed benefits of RD analysis over simulation.

6.1. Simulation Methodology

We implemented a cache simulator that models a cache hierarchy similar to the one in Figure 1. Our simulator uses the same PIN tool from Section 4 to generate parallel address traces, but instead of feeding these address traces into LRU stacks, our cache simulator replaces the LRU stacks with data cache models. In addition, it incorporates a directory cache model as well. Hence, our simulator can track the actual data cache miss and directory cache access counts as well as the actual contents of the directory cache (it simulates directory cache evictions) against which we can compare our profiler's predictions.

The rest of this article uses our simulator to conduct several experiments assuming different cache hierarchy configurations. For the validation experiments presented in this section, we always configure the simulator with private data caches–*i.e.*, we omit the shared cache shown in Figure 1. (Section 7.4 will conduct experiments with a shared cache). In particular, we simulate three levels of private cache–an L1, L2, and L3 per core. Coherence is enforced between the private caches using a directory-based MESI protocol. Also, we maintain inclusion across all caching levels, and all of the caches employ 64-byte blocks.

Along with the private data caches, our validation experiments also configure the cache simulator with a Cuckoo directory [Ferdman et al. 2011]. Cuckoo uses multiple hash functions and iterative re-insertion to increase the effective associativity of the directory cache. (We limit re-insertion attempts to a maximum of 32 in our simulator). This minimizes the over-provisioning needed to mitigate conflicts at the expense of more costly insertions. In our Cuckoo directory, we assume full-map directory entries which mirror the precise tracking of sharers in our profiler. Although we use full-map for validation, Section 7 will consider other directory entry organizations as well.

In our validation experiments, we vary both core count and cache capacity. Across the core count dimension, we perform simulations at three different configurations–16, 64, and 256 cores. (When varying core count, we keep the last-level cache size fixed at 64 MB total using 4 MB, 1 MB, and 256 KB per-core L3s, respectively). Across the cache capacity dimension, we perform simulations at four different last-level cache sizes–256 KB, 512 KB, 1 MB, and 2 MB per core. (When varying cache capacity, we keep core count fixed at 64 cores). The top half of Table VII specifies these and other data cache hierarchy parameters used for the validation experiments.

Core Counts	
16, 64, 256	
Private Data Cache Sizes (Associativities)	
Private L1: 16 KB (4-way)	
Private L2: 64 KB (8-way)	
Private L3: 256 KB, 512 KB, 1 MB, or 2 MB (all 8-way)	
Directory Cache Coverages (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)

Table VII: Core count and data / directory cache size parameters for the private data cache hierarchy modeled in the cache simulator.

Besides cache capacity, we also scale the Cuckoo directory's capacity along with the private data caches. For the validation experiments in this section, we always size the directory cache to maintain a 200% coverage over the private data caches. A Cuckoo directory can be much smaller than this, but we choose 200% coverage to virtually eliminate directory cache evictions due to conflicts. This allows us to validate the contents of the directory cache induced by the directory's access stream. The directory cache's contents is also affected by conflicts. We will assess the effects of conflicts later in Section 7.1 by varying the coverage. The bottom half of Table VII indicates all of the directory cache sizes we will consider, including the 200% coverage used in this section.

Lastly, for our validation experiments, we report the percent error between values predicted by our profiles and those measured in our simulator, computed as:

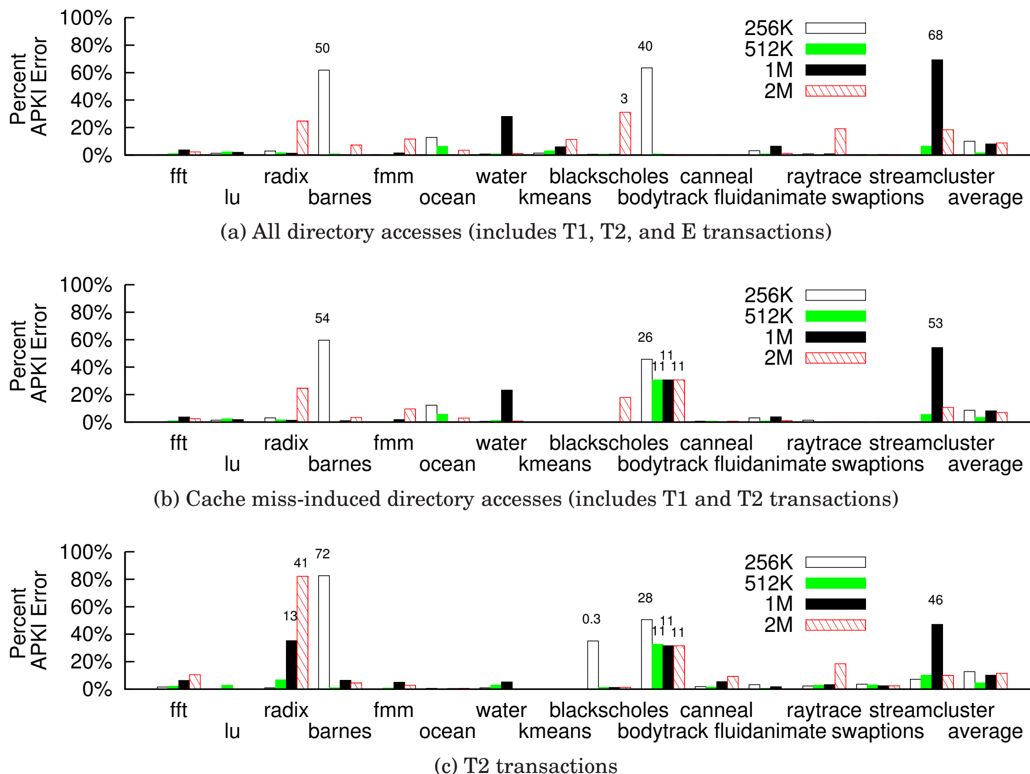
$$Percent\ Error = \frac{|\ Predicted - Simulated\ |}{Simulated} \tag{1}$$

In cases where the value in question is extremely small, Equation 1 can report a very large percent error even though the absolute error is miniscule. These cases are benign and occur only because of the error metric's instability (*i.e.*, a denominator that approaches zero). An alternate metric we use is "Offsetted Percent Error" in which we again use Equation 1, but first add an appropriately small offset to both the predicted and simulated values to mitigate the divide-by-zero problem.

6.2. Directory Access Count Errors

Figure 12 quantifies the error in our directory access results from Section 5.1. In the three graphs, we plot the percent error in directory APKI predicted by our profiles compared to simulation for each of our 15 benchmarks and the four simulated L3 cache sizes in Table VII. All results are for 64 cores. Figure 12(a) shows the percent error for all directory accesses (T1 + T2 + E transactions), Figure 12(b) shows the percent error for cache miss-induced directory accesses (T1 + T2 transactions), and Figure 12(c) shows the percent error for T2 accesses, corresponding to the curves labeled "With Notifications," "Total Misses," and "T2," respectively, from Figure 5.

As Figure 12 shows, the profiler's directory access predictions are very close to simulation most of the time, resulting in a small error on average. Across the 15 benchmarks, the error for all directory accesses, for cache miss-induced accesses, and for T2 accesses is 7.1%, 6.8%, and 9.7%, respectively, as shown by the "average" bars in Figure 12. The main reason for these errors is conflict misses that occur in the private



(a) All directory accesses (includes T1, T2, and E transactions)

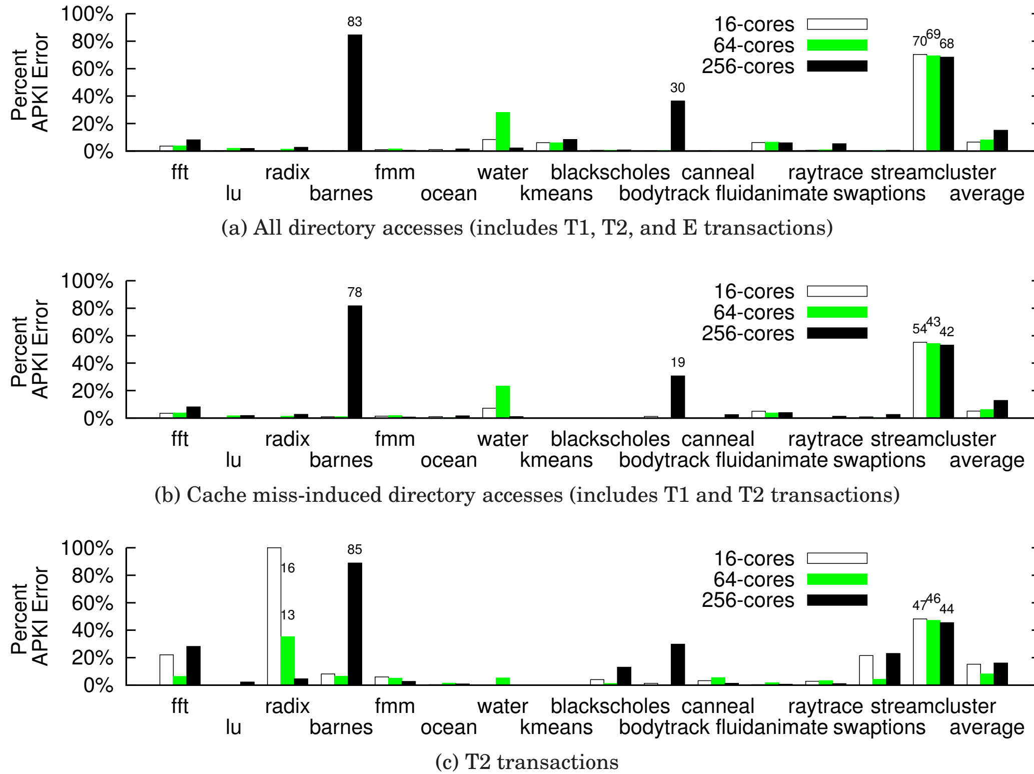
(b) Cache miss-induced directory accesses (includes T1 and T2 transactions)

(c) T2 transactions

Fig. 12: Percent APKI error at four different private cache sizes–256 KB, 512 KB, 1 MB, and 2 MB–for (a) all directory accesses, (b) cache miss-induced directory accesses, and (c) T2 transactions. Numbers above the bars report the offsetted percent error for the large-error cases.

data caches modeled by the simulator which our profiler does not account for. Consequently, our profiles under-predict the actual number of directory cache accesses. But as Figure 12 shows, not accounting for data cache conflicts causes a relatively small prediction error on average.

While average error is small, Figure 12 also shows there are a few cases with large error. In particular, 19 out of the 180 bars in Figure 12 exhibit greater than 30% error, with the worst-case bar exhibiting 82% error. But as mentioned in Section 6.1, some of these errors are benign: Equation 1 can report large error (even when the absolute error is miniscule) due to metric instability. To see how many cases this entails, we compute the offsetted percent error for the 19 large-error cases, and report them as numbers above their corresponding bars in Figure 12. (For directory access count, we use an offset value of 0.2 APKI). These results show the majority of large-error cases in Figure 12 are due to small APKI values that cause metric instability. Under the offsetted percent error metric, only 8 of the 180 bars in Figure 12 exhibit greater than 30% error, with the worst-case bar exhibiting 72% error. So, really only 4% of the cases are predicted poorly. For these cases, we find the large error is primarily due to steep drops in the access frequency curves (*e.g.*, Figure 5(o)). If the profiler slightly misjudges the capacity at which such drops occur, large errors may be reported around the



(a) All directory accesses (includes T1, T2, and E transactions)

(b) Cache miss-induced directory accesses (includes T1 and T2 transactions)

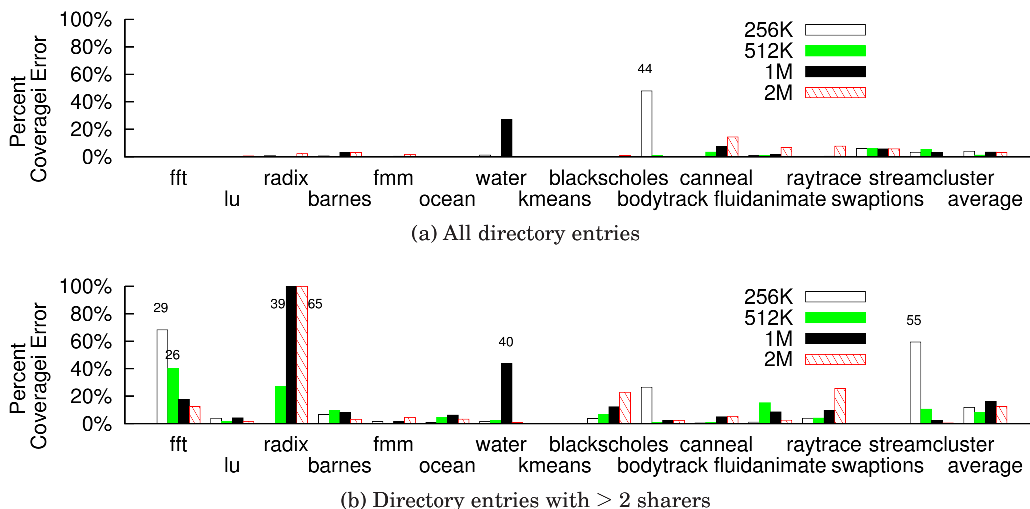
(c) T2 transactions

Fig. 13: Percent APKI error at three different core counts–16, 64, and 256–for (a) all directory accesses, (b) cache miss-induced directory accesses, and (c) T2 transactions. Numbers above the bars report the offsetted percent error for the large-error cases.

drop. This happens in barnes and streamcluster because their drops roughly coincide with one of the cache sizes we simulated.

In addition to validating the directory access counts for 64-core CPUs, we also validate across different core counts. Figure 13 shows these results. In the three graphs, we plot the percent error in directory APKI for each of our 15 benchmarks running on 16, 64, and 256 cores. For all core counts, we configured the cache simulator to model 64MB of total L3 cache. Figure 13(a) shows the error for all directory accesses, Figure 13(b) shows the error for cache miss-induced directory accesses, and Figure 13(c) shows the error for T2 accesses. And, numbers appearing above the bars report the offsetted percent error for cases exhibiting large errors ($> 30\%$).

The results in Figure 13 are similar to those in Figure 12. Again, the profiler's directory access predictions are very close to simulation most of the time, resulting in small errors on average (though slightly higher than those in Figure 12). Across the 15 benchmarks, the error for all directory accesses, for cache miss-induced accesses, and for T2 accesses is 9.9%, 8.0%, and 13.2%, respectively, as shown by the "average" bars in Figure 13. As with cache size scaling, a few of the datapoints have large errors: 16 out of the 135 bars in Figure 13 exhibit greater than 30% error, with the worst-case bar exhibiting 98% error. Under the offsetted percent error metric, 12 of the 135 bars in Figure 13 exhibit greater than 30% error, with the worst-case bar exhibiting 85% error. So, we find 9% of the cases are predicted poorly (again, a little worse than Figure 12).



(a) All directory entries

(b) Directory entries with ≥ 2 sharers

Fig. 14: Percent coverage error at four different private cache sizes–256 KB, 512 KB, 1 MB, and 2 MB–for (a) all directory entries, and (b) entries with 2 or more sharers. Numbers above the bars report the offsetted percent error for the large-error cases.

But overall, Figures 12 and 13 show our profiler can predict directory cache accesses accurately for the large majority of cases.

6.3. Directory Coverage Errors

Figure 14 quantifies the error in our coverage results from Section 5.2. To acquire these results, we ran our data and directory cache simulator using a Cuckoo directory with 200% coverage, and measured the average number of live directory entries in the directory cache. (As explained in Section 6.1, 200% coverage ensures virtually no entries are evicted due to conflicts, allowing us to validate the contents of the directory cache in the absence of conflicts). The graphs in Figure 14 plot the percent error in coverage predicted by our profiles compared to simulation for each of our 15 benchmarks and the four simulated L3 cache sizes in Table VII. All results are for 64 cores. Figure 14(a) shows the coverage error for all directory entries, and Figure 14(b) shows the coverage error for entries with 2 or more sharers, corresponding to the curves labeled "Total" and "$\geq$ 2 sharers," respectively, from Figure 7. Similar to Section 6.2, we also compute offsetted percent error for the large-error cases ($>$ 30%), and report them as numbers appearing above the corresponding bars. (For coverage, we use an offset value of 1%).

As Figure 14 shows, the profiler's directory coverage predictions are very close to simulation in most cases, resulting in a small error on average. Across all 15 benchmarks, the error for the total coverage is only 2.9%, while the error for coverage from directory entries with 2 or more sharers is 12.1%, as shown by the "average" bars in Figure 14. The main reason for these errors is the directory access errors already discussed in Section 6.2. Because our profiler does not correctly predict all directory cache accesses, it also does not perfectly track the directory's contents.

Similar to the results from Section 6.2, there are a few cases in Figure 14 with large error. In particular, 7 out of the 120 bars in Figure 14 exhibit greater than 30% error, with the worst-case bar exhibiting 101% error. But as in Figures 12 and 13, the instability of the percent error metric causes some of these large-error cases. Under the offsetted percent error metric, 5 out of the 120 bars in Figure 14 exhibit greater

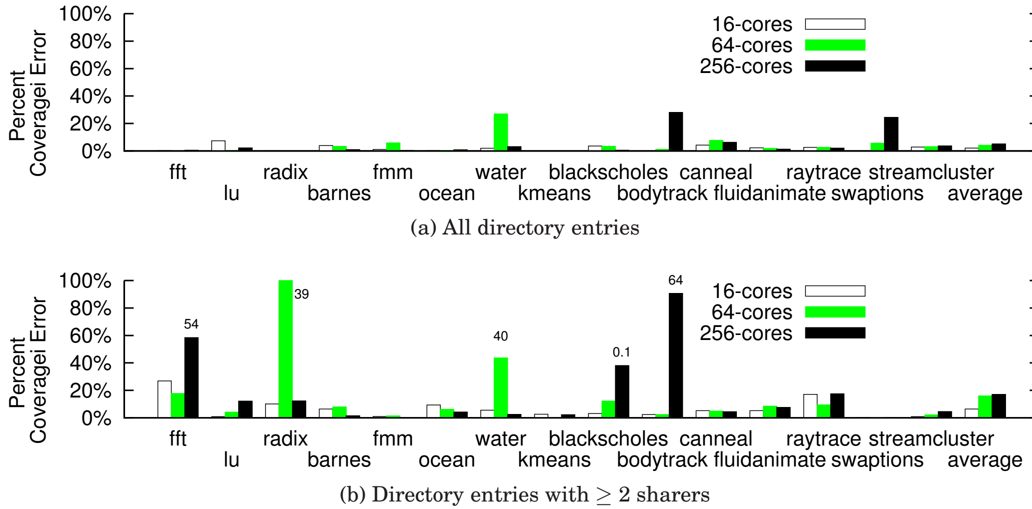


Fig. 15: Percent coverage error at three different core counts–16, 64, and 256–for (a) all directory entries, and (b) entries with 2 or more sharers. Numbers above the bars report the offsetted percent error for the large-error cases.

than 30% error, with the worst-case bar exhibiting 65% error. So, we find 4% of the cases are predicted poorly.

Figure 15 shows validation results at different core counts. Similar to Figure 13, we plot the percent error in directory coverage for each of our 15 benchmarks running on 16, 64, and 256 cores with 64 MB of total L3 cache. Figure 15(a) shows the error for all directory entries, and Figure 15(b) shows the error for entries with 2 or more sharers. And, numbers appearing above the bars report the offsetted percent error for cases exhibiting large errors ($> 30\%$). The errors in Figure 15 are similar to those in Figure 14. Average error is reasonable–3.7% and 13.2% for all directory entries and entries with 2 or more sharers, respectively. And again, a few cases have large errors: 5 out of the 90 bars in Figure 15 exhibit greater than 30% error, with the worst-case bar exhibiting 99% error. Under the offsetted percent error metric, 4 out of the 90 bars exhibit greater than 30% error, with the worst-case bar exhibiting 64% error. So again, we find 4% of the cases are predicted poorly. But overall, Figures 14 and 15 show our profiler can predict directory coverage accurately for the large majority of cases.

6.4. Directory Access Distribution Errors

As discussed in Section 5.3, the “Total” and “≥ 2 accesses” curves in Figure 10 are identical to the “Total” and “≥ 2 sharers” curves in Figure 7 (because directory entries for private cache blocks are accessed exactly once–by a T1 transaction that fills the entry into the directory cache). So, the experiments from Section 6.3 that validate accuracy of the coverage breakdown between total and multi-sharer directory entries in Figure 7 also validate accuracy of the coverage breakdown between total and multi-access directory entries in Figure 10. In this section, we further validate access distribution results from Section 5.3, in particular for the temporal reuse results in Table V.

Figure 16 shows these validation results. In Figure 16, we plot the percent error in the percentage of directory entries that receive 3 or more directory accesses predicted by our profiles compared to simulation for each of our 15 benchmarks and the four simulated L3 cache sizes in Table VII. (The bars for 256 KB and 1 MB cache sizes in

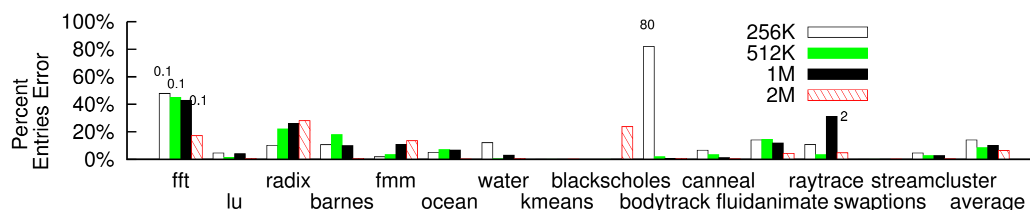


Fig. 16: Percent entries error for $\geq$ 3-access entries at four different private cache sizes–256 KB, 512 KB, 1 MB, and 2 MB. Numbers above the bars report the offsetted percent error for the large-error cases.

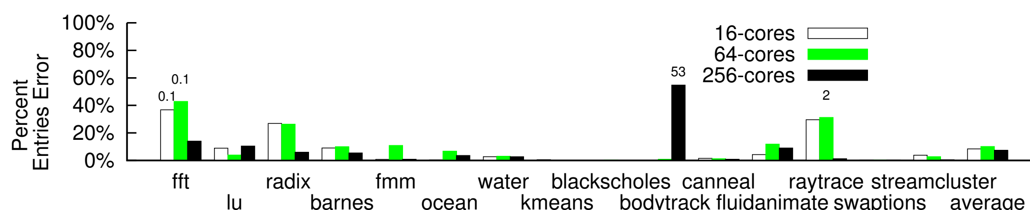


Fig. 17: Percent entries error for $\geq$ 3-access entries at three different core counts–16, 64, and 256. Numbers above the bars report the offsetted percent error for the large-error cases.

Figure 16 specifically validate the results from columns 5 and 6 of Table V). All of the results in Figure 16 are for 64 cores. In Figure 17, we perform the same validation experiments, except at different core counts–16, 64, and 256. When varying the core count, we keep the total L3 cache size fixed at 64 MB. Besides reporting percent error, we also compute the offsetted percent error metric for the large-error cases ($>$ 30%) and report them as numbers appearing above the corresponding bars. (As in Section 6.3, we again use an offset value of 1% for coverage).

As Figures 16 and 17 show, the profiler's prediction of the percentage of $\geq$ 3-access directory entries is very close to simulation in most cases, resulting in a small error on average. Across all 15 benchmarks, the error for the four simulated L3 cache sizes is 9.7% while the error for the three simulated core counts is 8.6%, as shown by the "average" bars in Figures 16 and 17, respectively. And similar to Figures 12–15, a few cases in Figures 16 and 17 have large error. In particular, 5 out of the 60 bars in Figure 16 and 4 out of the 45 bars in Figure 17 exhibit greater than 30% error, with the worst-case bars exhibiting 81% and 54% error, respectively. Under the offsetted percent error metric, only 1 out of the 60 bars in Figure 16 and 1 out of the 45 bars in Figure 17 exhibit greater than 30% error. So, just 2% of the cases are predicted poorly. Overall, Figures 16 and 17 show our profiler can accurately predict the percentage of $\geq$ 3-access directory entries in the large majority of cases.

6.5. Discussion: Timing-Related Errors

All of the experiments in Sections 6.2–6.4 are based on cache simulation. As such, they do not directly reflect the accuracy of our profile predictions compared to a timing-accurate simulator. The potential problem is that the data caches in a timing-accurate simulator would see a different memory reference stream as compared to the one generated by PIN. (Our PIN tool simply interleaves cores' memory reference streams in a round-robin fashion, which is different from the interleaving that would occur on a timing-accurate model). So, the data caches' miss behavior could be different, which

in turn could alter the directory cache behavior, resulting in errors that our validation experiments do not account for.

However, researchers have observed that for programs with homogeneous threads (the assumption made in our work), timing tends to affect *all threads equally*, thus it does not perturb inter-thread interleaving significantly compared to the round-robin ordering. In fact, it has been shown by Wu and Yeung [Wu and Yeung 2013] that PRD profiles acquired on a PIN profiler are indistinguishable from those acquired on a timing-accurate simulator. (Section 6.3 of the cited paper reports PRD values from PIN and timing-accurate profiles are within 99.9% of each other, averaged across all cache sizes). This suggests that the data caches' miss behavior on a timing-accurate simulator should be very close to what we observe on our PIN-based simulator. (Indeed, Section 6.4 of Wu and Yeung's paper uses PRD profiles to predict data cache misses–in essence, the directory access stream that we study in Section 6.2–in a timing-accurate simulator. Using a subset of the benchmarks we use in this article, they report a prediction accuracy of 8.5%, which is consistent with the accuracy we observe in Section 6.2.) Because the directory cache is directly driven by the data cache-miss stream, this provides some confidence that the errors reported in Sections 6.2–6.4 are representative of what one would observe for a timing-accurate simulator.

6.6. Profiler Speed

In this section, we compare the speed of our profiler against the speed of the simulator to quantify the potential evaluation speed benefits of RD analysis. Table VIII presents our results. In Table VIII, the 2nd and 3rd columns (labeled "SC" and "PC") report the number of total configurations acquired by the simulator and profiler, respectively, for the 64-core results obtained in this article. The 4th column of Table VIII (labeled "P/S") reports their ratio–*i.e.*, how many times more samples were acquired by the profiler compared to the simulator. Then, the next two columns of Table VIII (labeled "ST" and "PT") report the profiler and simulator's running time, respectively, in seconds. And, the last column of Table VIII reports the speedup of the profiler over the simulator in terms of time to acquire each configuration. Finally, the last two rows of Table VIII report the average of the "P/S" and "Speedup" columns as well as the total sum of the "ST" and "PT" columns.

Table VIII shows that the total time spent running the profiler to obtain the results for this article was quite a bit higher than the time spent running the simulator. Comparing the total running times for the profiler and simulator (1,342,499 seconds versus 308,495 seconds), Table VIII shows the profiler took 4.35x more time to run. However, the profiler evaluated far more configurations. While we ran only 4 different cache capacities per benchmark on the simulator, the profiler evaluated $>$ 100 different cache capacities in almost all the benchmarks. (The profiler maintains statistics at 16KB increments, so the number of evaluated configurations depends on the maximum reuse distance value, which varies across benchmarks). In most benchmarks, the profiler evaluated between 30-75x more configurations than the simulator. Averaged across all benchmarks, it evaluated 55.8x more configurations. Thus, in terms of running time per evaluated configuration, the profiler is much faster than the simulator, between 2.9x and 28.8x faster as shown in the last column of Table VIII. On average, the profiler is 12.0x faster than the simulator.

While Table VIII shows a significant speedup for the profiler, the actual speedups would be even higher if one considers the amount of information tracked. The simulator did not validate *every* statistic from the profiler; it only validated the main results from Figures 5, 7, and 10. So, we streamlined its implementation to only track the necessary statistics. In contrast, the profiler keeps more information than the simulator to facilitate the detailed studies in Section 5–for example, keeping coverage statistics

Benchmark	SC	PC	P/S	ST	PT	Speedup
fft (kernel)	4	274	68.5	5420.5	22708.4	16.4
lu (kernel)	4	300	75	54474.6	148253.0	27.6
radix (kernel)	4	300	75	8599.3	69004.7	9.3
barnes	4	271	67.8	129821.0	409007.0	21.5
fmm	4	300	75	9847.3	78386.5	9.4
ocean	4	260	65	3404.2	26708.2	8.3
water	4	101	25.3	4724.2	17333.4	6.9
kmeans	4	300	75	7891.8	58117.3	10.2
blackscholes	4	129	32.3	5234.3	16229.9	10.4
bodytrack	4	36	9	14375.9	43950.5	2.9
canneal	4	147	36.8	316.7	2004.8	5.8
fluidanimate	4	145	36.3	5212.6	25012.8	7.6
raytrace	4	182	45.5	27607.6	43602.9	28.8
swaptions	4	300	75	8497.4	64717.4	9.8
streamcluster	4	300	75	23067.6	317462.0	5.4
Average			55.8			12.0
Total Sum				308495	1342499	

Table VIII: Number of simulated (SC) and profiled (PC) configurations, and the ratio of profiled to simulated configurations (P/S). Simulation (ST) and profiler (PT) time in seconds, and profiler speedup per configuration.

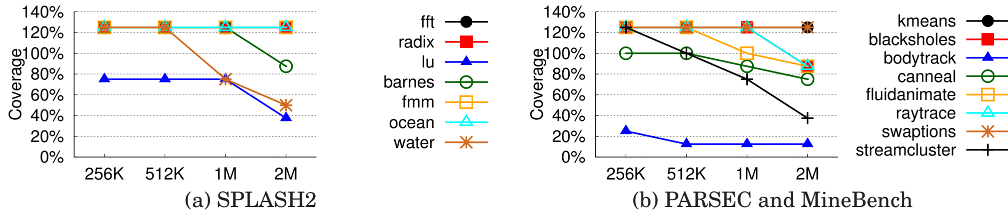
by sharing degree (Figure 8 and Table IV) and tracking intrinsic coherence transactions (Table III), among other statistics. If we were to eliminate these and perform an "apples to apples" comparison, the profiler would exhibit even greater speedups.

7. CASE STUDIES

Having presented our insights on directory cache behavior and validated their accuracy, we now explore their implications for the effectiveness of existing directory techniques. This is done through several case studies. First, Section 7.1 studies the implications of our coverage insights on the size of Cuckoo directories. Then, Sections 7.2 and 7.3 study the implications of our sharing insights on the capacity scaling of SCD and DGD directories. Next, Section 7.4 studies the implications of our access distribution insights on multi-level directory techniques. Finally, Section 7.5 provides qualitative discussion on the implications of our directory access count insights on directory techniques in general.

7.1. Cuckoo

The directory cache's capacity is determined by two factors: the number of directory entries stored in the directory cache, and the size of each of those entries. Section 5.2 provides insights with implications for the former. It shows coverage can reduce significantly with CPU scaling, especially data cache size scaling. But the magnitude of the reductions are application dependent. For applications with large drops in coverage, the implication is that the number of unique directory entries in the directory cache, as a percentage of the number of on-chip cache blocks, can potentially be reduced as data cache capacity scales. In other words, the number of directory entries can grow slower than the rate at which the data caches are scaled. However, it is unclear whether this translates into actual benefits for directory caches. One issue is our coverage results do not account for practical considerations like the eviction policies that real directory caches employ and the resulting conflicts that can occur. Another issue is the application-dependent magnitude of the reductions. In this section, we con-



(a) SPLASH2 (b) PARSEC and MineBench

Fig. 18: Minimum Cuckoo directory size (in terms of coverage) for 1% eviction rate as a function of private data cache size.

duct experiments on real directory caches to verify the actual capacity reductions, and discuss how these potential capacity savings can be exploited.

7.1.1. Methodology. We perform experiments using Cuckoo directories [Ferdman et al. 2011]. Directory caches often employ over-provisioning (beyond the number of unique data cache blocks in the CPU's private caches) to keep spurious evictions–*i.e.*, from conflicts in the directory cache–at a minimum. As discussed in Section 6.1, Cuckcoo directories use sophisticated hashing and re-insertion techniques to increase their effective associativity, thus minimizing the amount of over-provisioning needed. By assuming Cuckoo, our results reflect directory cache scaling given state-of-the-art conflict-mitigation techniques.

Our experiments are conducted on the same 3-level data cache + directory cache simulator from Section 6.1. In our experiments, we simulate 64-core CPUs that employ the data cache sizes (and associativities) listed in Table VII, including the four L3 data cache sizes. However, instead of setting the number of Cuckoo directory entries to 200% coverage as was done in Section 6, we simulated all of the coverage values listed in the bottom portion of Table VII.[3] After simulating these coverage values, we identified the minimum capacity at which Cuckoo can still effectively cache all of the required directory entries. In particular, we looked for the smallest Cuckoo directory for which 1% (or fewer) of the directory entry insertions result in eviction of a live directory entry. Notice, while a 1% eviction rate will likely lead to high performance, it is important to point out that we do not actually know the impact of evicting live directory entries on end-to-end performance. (For all we know, even smaller Cuckoo directories with $>$ 1% eviction rates would still result in high performance). Unfortunately, because our profiler does not model timing, we cannot analyze the exact performance impact. Doing so would require detailed architectural simulation.

Lastly, as in Section 6 (as well as the original Cuckoo paper [Ferdman et al. 2011]), we assume full-map directory entries in all of our Cuckoo experiments. Sections 7.2 and 7.3 will study the effect of more advanced directory entry organizations. In this section, we focus on how the number of unique directory entries scales.

7.1.2. Results. Figure 18 presents our Cuckoo results. In Figure 18, we plot the minimum number of Cuckoo directory entries (in terms of coverage) that achieve a 1% eviction rate during insertions along the Y-axis as a function of the four L3 data cache sizes we simulated along the X-axis. This is done for all 15 of our benchmarks. Figure 18(a) presents the results for the SPLASH2 benchmarks, while Figure 18(b) presents the results for the PARSEC and MineBench benchmarks.

[3]Different numbers of memory arrays, *i.e.* "ways" in Table VII, are used to maintain a power-of-2 number of sets as coverage is varied.

Figure 18 confirms that the number of Cuckoo directory entries–in terms of coverage–drops with data cache size scaling for many benchmarks. Most benchmarks require 125% coverage at 256KB private caches. But as private caches scale to 2MB, only six benchmarks remain at 125% coverage while nine benchmarks drop in coverage. In particular, five benchmarks drop to 75–87.5% coverage, while four benchmarks drop to 50% coverage or less.

In addition, comparing Figures 18 and 7, we see the minimum number of Cuckoo directory entries is correlated to the profile-based coverage. In most cases, the number of Cuckoo directory entries required is 30–50% higher than the profiled coverage. This represents the amount of over-provisioning needed in the directory cache to mitigate conflicts.[4] These results show that 9 out of 15 of our benchmarks can tolerate scaling the number of Cuckoo directory entries at a slower rate than scaling of the data cache size. Moreover, our analysis from Section 5.2 can help identify the minimum Cuckoo directory at any data cache capacity.

Although Figure 18 demonstrates potential benefits, whether they can be exploited in real systems is an open question. The problem is the coverage trends are application dependent. Since general-purpose systems are designed for worst-case behavior, a fixed coverage of 125% would be needed to support all of our benchmarks since a few of them do not exhibit coverage reductions in Figure 18. A promising idea, perhaps, is dynamic resizing. Much like data cache resizing techniques [Albonesi 1999; Madan et al. 2009; Malik et al. 2000; Powell et al. 2000; Yang et al. 2002], the directory cache could be resized at runtime, for example at the beginning of a program's run, enabling power gating of unused portions. (To our knowledge, there have not been directory resizing techniques studied in the literature). Besides runtime techniques, application-specific architectures could also exploit the potential benefits in Figure 18 at design time.

7.2. SCD

To improve the directory's capacity scaling, researchers have investigated reducing the size of directory entries. As discussed in Section 1, numerous techniques have been explored that provide compact directory entry formats. The efficacy of these techniques depends in large part on applications' sharing patterns. Hence, the results from Section 5.2, which show how on-chip sharing patterns change with scaling, can provide significant insights into the behavior of these previous techniques.

In Section 5.2, our profiler only tracked the number of sharers per directory entry. However, it is straight-forward to extend the profiler to acquire additional information for modeling specific directory entry formats. In particular, this section uses our profiler to study the SCD technique [Sanchez and Kozyrakis 2012]. As in Section 7.1, our analyses focus on data cache capacity scaling (rather than core count scaling) since this is the dimension across which on-chip sharing patterns exhibit the greatest variation.

7.2.1. Methodology. SCD employs two different directory entry formats to track cache blocks with both narrow and wide sharing patterns efficiently. For narrow sharing, SCD provides a single limited pointer format capable of tracking up to three sharers. When a cache block overflows the limited format, SCD switches to a multi-entry hierarchical format. In this case, the original limited directory entry becomes a *root entry* that tracks groups of cores sharing the cache block, while multiple *leaf entries* are allocated to track individual sharers within each group. In particular, each leaf entry

[4]The minimum over-provisioning is actually even less because the smallest Cuckoo directory with a 1% eviction rate during insertions is usually in between two coverage values we simulate, in which case we pick the larger value.

corresponds to a single group of cores and contains the portion of a full-map bit vector for that group.

This approach enables compact directory entries. For example, given a 256-core CPU, SCD's limited format stores three 8-bit pointers. And, its hierarchical format, which sub-divides the CPU into 16 groups of 16 cores each, consists of root entries with 16-bit vectors (one bit per group) and leaf entries with 16-bit portions of a full-map vector. Each of these formats (limited, root, and leaf) can fit in a 32-bit directory entry, allowing SCD to provide an 87.5% reduction in directory entry size compared to a 256-bit full-map entry. At the same time, though, SCD also increases the total number of directory entries since the hierarchical format allocates root and leaf entries. Granted, only leaf entries associated with groups of cores with at least one sharer need to be allocated. Nevertheless, whether or not SCD provides an overall capacity scaling benefit depends on applications' sharing patterns, and on how sharers are distributed across groups of cores.

We modified our profiler to study SCD scaling for a 256-core CPU. In addition to the "dir entry sharer ctrs" in Figure 4, we also maintain a set of SCD entry counters. Just like the sharer counters, we maintain one SCD entry counter per unique data block contained in all of the LRU stacks at every capacity, CS_i. These counters track the number of live directory entries allocated per cache block within an SCD directory for different private cache sizes. When the number of sharers for a cache block at a capacity CS_i is three or less, the corresponding SCD entry counter is set to 1. However, when the number of sharers exceeds three, the counter is increased based on the sharing pattern. In particular, for every 16-core group with at least one sharer, the counter is incremented by 1.

Notice, because the encoding of sharing via hierarchical directories depends on the distribution of sharers across leaf entries, the utilization of SCD directories is affected by the mapping of threads to cores. In our experiments, we assume the mapping is based on thread ID–*i.e.*, thread 0 is mapped to core 0, thread 1 is mapped to core 1, and so on. This is the same thread-to-core mapping used in the SCD paper [Sanchez and Kozyrakis 2012].[5] We did not try other mappings, so our experiments only study the behavior for the original SCD implementation, which is our main goal.

7.2.2. Results. Figure 19 presents our SCD results. In Figure 19, the dotted lines labeled "SCD" plot the number of SCD directory entries in terms of coverage as a function of private data cache size. For comparison, the solid lines labeled "Total" plot the number of directory entries from our baseline profiler–*i.e.*, assuming a single directory entry for each unique data cache block. And, the dashed lines labeled "$\geq$ 2 sharers" plot the number of directory entries with 2 or more sharers, such that the gap between the "Total" and "$\geq$ 2 sharers" curves breaks down the coverage for single-sharer entries. (The "Total" and "$\geq$ 2 sharers" curves in Figure 19 correspond to the curves with the same labels in Figure 7, and reflect the coverage for a full-map directory). All results are for 256-core CPUs.

Each benchmark in Figure 19 falls into one of three categories depending on SCD's coverage relative to full-map. Fft, radix, ocean, kmeans, blackscholes, raytrace, and swaptions form the first category. For these seven benchmarks, SCD never increases the number of directory entries compared to a full-map directory, so the "SCD" and "Total" curves are coincident. As discussed in Section 5.2, cache blocks with narrow sharing tend to dominate widely shared cache blocks. Among the benchmarks in this first category, the vast majority of cache blocks exhibit 3 or fewer sharers, so they fit in SCD's limited pointer format, with almost no cache blocks requiring the more

[5]The mapping used is not described in the paper, but we verified it with the authors.

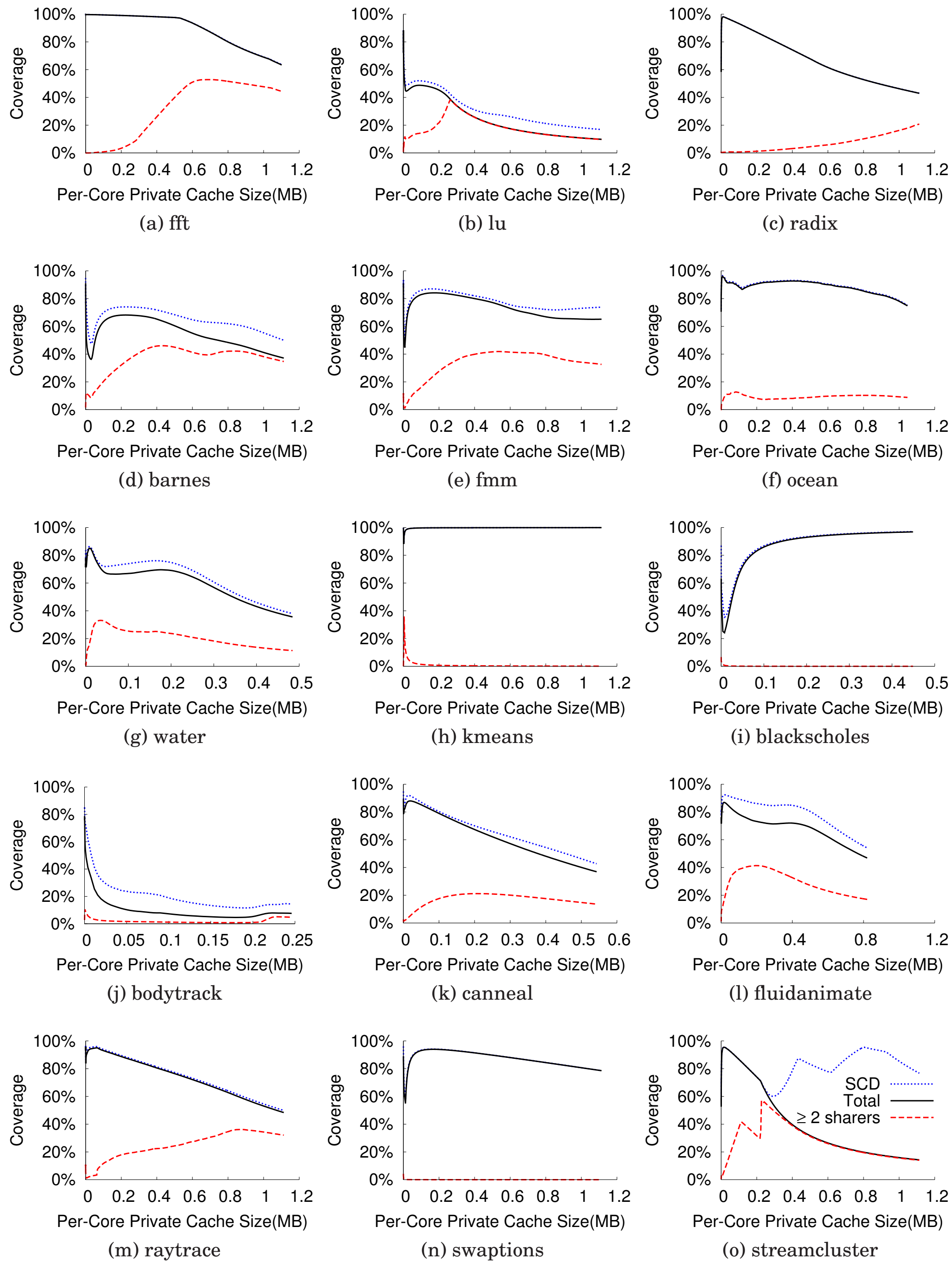


Fig. 19: Coverage for SCD directory entries *vs.* private data cache size. The “Total” and “$\geq$ 2 sharers” curves from the baseline profiler (similar to the corresponding curves in Figure 7) are plotted for comparison. All results are for 256-core CPUs.

expensive multi-entry format. This is the best that SCD can ever do–*i.e.*, the "Total" curves are a lower-bound on coverage for SCD. In this case, SCD delivers the greatest reduction in directory cache capacity since the per-entry savings (87.5%, as mentioned in Section 7.2.1) are not offset by any increase in the number of directory entries.

Lu, barnes, fmm, water, bodytrack, canneal, and fluidanimate form the second category. For these seven benchmarks, SCD increases the number of directory entries slightly compared to a full-map directory, as indicated by the gap between the "SCD" and "Total" curves. As discussed in Section 5.2, data cache size scaling causes application-level sharing to manifest on chip, exposing it to the directory. Among the benchmarks in this second category, the increased sharing causes some cache blocks to exhibit more than 3 sharers, so SCD creates a number of multi-level directory entries which pushes the "SCD" curves above the "Total" curves in Figure 19. In many cases, the gap grows with increased sharing. For example, in lu and barnes, the "SCD"–"Total" gaps are correlated to the rise in the "≥ 2 sharers" curves (and drop in gaps between the "Total" and "≥ 2 sharers" curves). But the number of additional multi-level entries is small. So, like the first category, SCD still provides a significant reduction in directory cache capacity for the benchmarks in the second category.

Finally, streamcluster forms the third category. For this benchmark, SCD increases the number of directory entries significantly compared to a full-map directory, but the behavior is capacity dependent. Because our framework can analyze the full spectrum of cache sizes, Figure 19(o) reveals the behavior. At small cache sizes, most cache blocks are private, and fit in the limited pointer format. But as cache size increases, the number of sharers also increases, and by 256 KB, almost all cache blocks exhibit multiple sharers. (As Figure 19(o) shows, the gap between the "Total" and "≥ 2 sharers" curves disappears, indicating there are no single-sharer directory entries). Unfortunately, most cache blocks are shared by more than three cores, with 4–10 being typical. Also, *the sharing pattern is random*. Hence, not only do most cache blocks require the multi-entry format, but they generate a large number of leaf entries owing to the random distribution of sharers across the chip.

7.3. DGD

Another way to improve the directory cache's scaling is by coalescing similar directory entries. It is well known that privately accessed data can dominate shared data. Moreover, spatially contiguous regions of private data are often accessed by the same core. Hence, their associated directory entries can be coalesced into a single directory entry representing the entire region. DGD [Zebchuk et al. 2013] is an example of a state-of-the-art technique that exploits such contiguous private regions. Like SCD, the efficacy of DGD depends on applications' sharing patterns, so once again, the results from Section 5.2 can provide insights into the behavior of DGD. In this section, we modify our profiler to study a DGD directory.

7.3.1. Methodology. DGD employs two different directory entry formats to enable efficient tracking of contiguous and privately owned data regions. In particular, a *region entry* in DGD is a single directory entry that can track multiple consecutive cache blocks if they are all privately accessed by the same core–*i.e.*, the region owner. When a core different from the region owner accesses a cache block within the region (either a block that becomes shared or a block accessed for the first time by a non-region owner), then a *block entry* is created. Each block entry contains a traditional full-map sharer vector so that it can track any sharing pattern. Meanwhile, the region entry must be changed to reflect the region's mixed sharing status. To do this, region entries contain *presence vectors* with one bit per cache block in the region. Individual presence bits can be set to indicate the cache blocks that are private to the region owner, or

cleared for those cache blocks that are no longer private to the region owner but are instead tracked by block entries. Both region and block entries are stored in the same directory cache structure.

We modified our profiler to study DGD scaling for a 64-core CPU. Assuming 64-byte cache blocks, our modified profiler employs 4 KB regions, each comprising 64 contiguous cache blocks. (All regions are aligned to a 4 KB boundary). Hence, region entries contain 64-bit presence vectors while block entries contain 64-bit sharer vectors. To account for the region entries, our profiler maintains a set of DGD region counters, one per 4 KB region from which there is at least 1 cache block in the LRU stacks. Our modified profiler also maintains the same "dir entry sharer ctrs" from Figure 4 that were explained in Section 4. Like the sharer counters, there is one set of region counters for every cache capacity, CS_i, in Figure 4, accounting for the region entries that would be allocated at each capacity.

The DGD region counters are managed as follows. On an access to a cache block, the profiler searches for the corresponding region counter at every capacity CS_i above which the referenced block is promoted. If the region counter is not found at a particular capacity (*i.e.*, the first T1 transaction to the region for that cache capacity), then the profiler allocates a new region counter, sets its count to 1, and makes the requesting core the owner of the region. Subsequent misses by the region owner to other cache blocks in the region (*i.e.*, additional T1 transactions to the region) increment the region counter by 1. However, if a core other than the region owner suffers a cache miss to a cache block tracked by the region entry (*i.e.*, a T2 transaction), then the profiler decrements the region counter by 1. These are transactions that create a block entry for a cache block whose tracking was previously coalesced within the region. The region counter is also decremented with each directory notification for a cache block whose tracking was coalesced within the region. Finally, when the region counter decrements to zero, it is deallocated.

At any given moment in time, the number of region counters allocated in our profiler at a particular capacity, CS_i, is the number of region entries in the corresponding DGD directory. However, the number of allocated sharer counters over-counts the block entries. This is because sharer counters corresponding to cache blocks privately owned by a region owner would be coalesced in their region entry. To compute the number of block entries, we subtract the sum of all region counter values (*i.e.*, the total number of coalesced cache blocks) at a particular capacity, CS_i, from the number of allocated sharer counters.

7.3.2. Results. Figure 20 presents our DGD results. In Figure 20, the dotted lines labeled "DGD" plot the number of DGD directory entries (including both region and block entries) in terms of coverage as a function of private data cache size. For comparison, the solid lines labeled "Total" plot the number of directory entries from our baseline profiler–*i.e.*, assuming a single directory entry for each unique data cache block. And, the dashed lines labeled "$\geq$ 2 sharers" plot the number of directory entries with 2 or more sharers, such that the gap between the "Total" and "$\geq$ 2 sharers" curves breaks down the coverage for single-sharer entries. (The "Total" and "$\geq$ 2 sharers" curves in Figure 20 are identical to the corresponding curves in Figure 7). All results are for 64-core CPUs.

The first observation we make is that DGD's coverage always lies in between the "Total" and "$\geq$ 2 sharers" curves in Figure 20. This is because at worst, every unique cache block allocates a separate directory entry–either a block entry or a region entry. In this case, there is no coalescing, so DGD's coverage matches the "Total" curve. On the other hand, all privately accessed cache blocks could be coalesced. At best, the single-sharer directory entries that would have been allocated in the baseline profiler

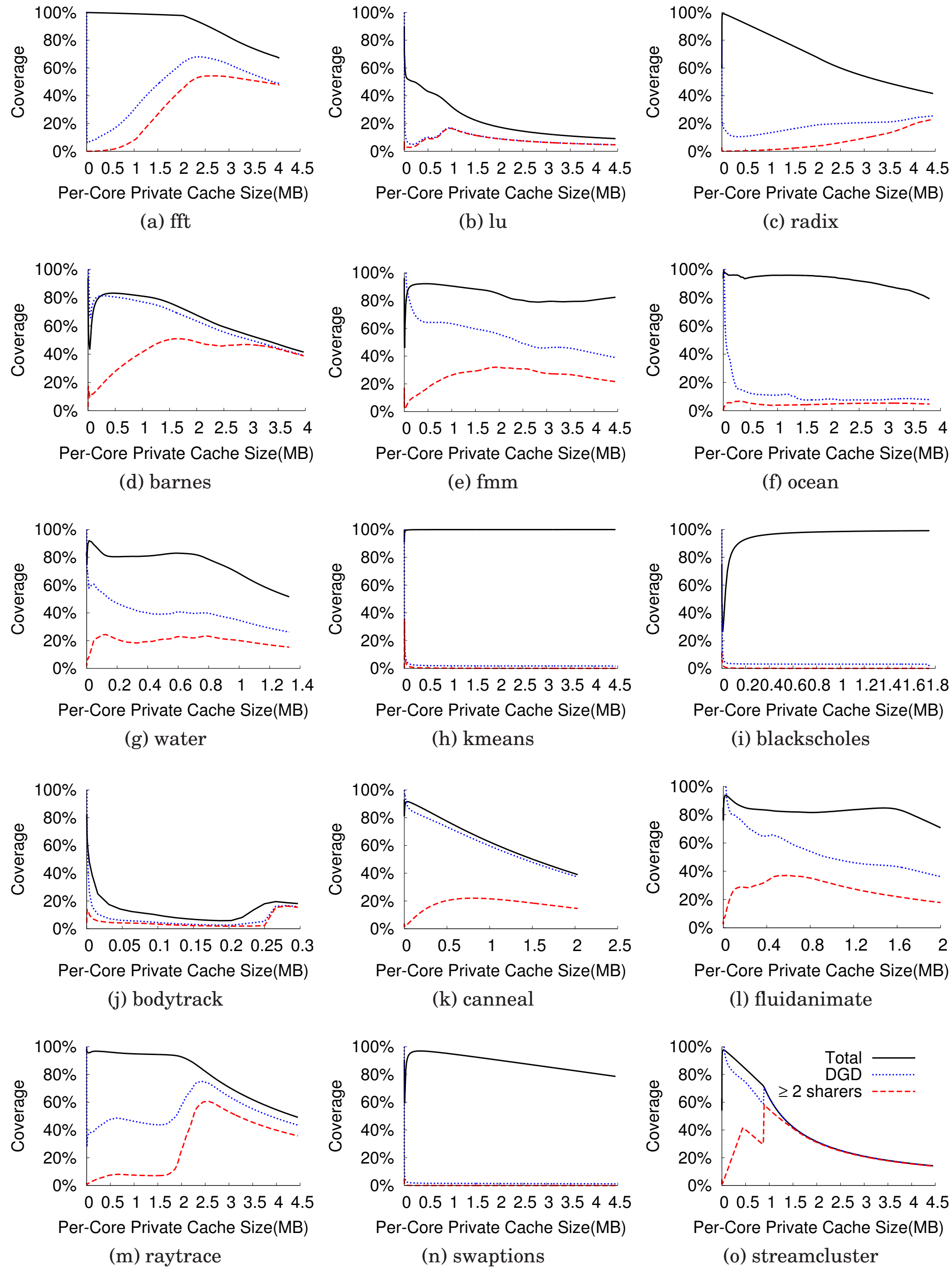


Fig. 20: Coverage for DGD directory entries *vs.* private data cache size. The "Total" and "$\geq$ 2 sharers" curves from the baseline profiler (identical to the corresponding curves in Figure 7) are plotted for comparison. All results are for 64-core CPUs.

(*i.e.*, the gap between the "Total" and "$\geq$ 2 sharers" curves) are maximally coalesced, reducing them by a factor 64x. In this case, DGD's coverage is slightly higher than the "$\geq$ 2 sharers" curve (shared cache blocks still need to allocate individual block entries), where the slight increase quantifies the number of region entries allocated to track the coalesced private cache blocks. So, Figure 20 shows our profiler can provide both the upper and lower bounds on DGD coverage at every private cache capacity.

This provides insights into the efficacy of the DGD technique. In particular, because the "Total" curves in Figure 20 are an upper bound, DGD experiences the same coverage reductions visible in the "Total" curves that come from increased sharing with capacity scaling, as shown in Section 5.2. However, this is not a benefit that we can credit to DGD as all directory techniques already benefit from this. Instead, the real benefit of DGD lies in *reducing the gap* between the "Total" and "$\geq$ 2 sharers" curves which comes from coalescing private entries. When interpreting Figure 20 with this observation in mind, we see that at small cache sizes, DGD noticeably reduces coverage for most benchmarks (all but barnes, canneal, fluidanimate, and streamcluster). But as cache size increases, DGD's benefit becomes more limited.

DGD's reduced efficacy at large cache sizes stems from two reasons. First, the gap between the "Total" and "$\geq$ 2 sharers" curves itself reduces with capacity scaling. This effect is severe in lu, barnes, bodytrack, raytrace, and streamcluster where after a certain capacity the gap essentially disappears. It is also pronounced in fft, radix, water, and canneal. As discussed in Section 5.2, cache size scaling not only increases sharing, but it also reduces the number of privately accessed cache blocks. This eliminates much of the headroom for DGD to reduce coverage through coalescing in most of the benchmarks. And second, at large cache sizes, DGD is also unable to coalesce all of the private directory entries for several benchmarks, leaving some potential coverage reduction on the table. This occurs in fmm, water, canneal, fluidanimate, and raytrace. Only for four benchmarks–ocean, kmeans, blackscholes, and swaptions–is DGD able to significantly reduce coverage at large cache sizes as well as small cache sizes.

7.4. Multi-Level Directories

While SCD and DGD try to compact the directory's contents, yet another approach for improving the directory's capacity scaling is to exploit asymmetric access distribution across directory entries. Multi-level directories are an example of such an approach. In a multi-level directory, the directory cache is split into two separate structures: a small *L1 directory cache* that is backed by a larger *L2 directory cache*. The idea is to store the most frequently accessed directory entries in the L1 where they can be accessed with low latency. Then, as directory entries are evicted from the L1, they are placed in the L2 rather than being discarded. Even though the L2 incurs higher latency, retaining the directory entries saves from having to evict the associated data cache blocks, or to re-probe caches (via broadcast) when subsequent coherence operations are needed. (The former increases data cache misses and degrades performance whereas the latter consumes on-chip bandwidth and power consumption for broadcasting, which can be quite significant given the large core counts we consider). Because its performance requirements are less stringent, the L2 can be implemented using slower but much denser storage. By using denser memories for the L2 directory cache, the overall cost for capacity scaling can be reduced.

For example, Figure 21 illustrates a multi-level directory design that we consider in this section. To achieve low latency, the L1 directory cache is implemented in SRAM on the same chip as the cores, similar to a traditional monolithic directory cache. But, the L2 directory cache in Figure 21 is implemented in denser (though slower) stacked DRAM. Other implementations have also been proposed in the literature. In particular, PS-Dir [Valls et al. 2012] keeps the L2 directory on the same chip as the cores, but

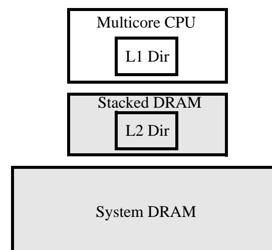


Fig. 21: Example multi-level directory cache.

implements it in eDRAM to reduce its on-chip footprint. WayPoint [Kelm et al. 2010], however, moves the L2 directory off-chip into system DRAM.

The access distribution insights from Section 5.3 has implications for such multi-level directories. In particular, Section 5.3 shows there is high temporal reuse of certain directory entries, especially during T2 transactions. This implies that high L1 directory cache hit rates are possible. But it is not 100% certain that our insights will translate into actual benefits. One significant issue is Section 5.3 does not account for the management policies that real multi-level directory caches would employ. In this section, we conduct experiments on an actual multi-level directory model to explore the implications of our insights from Section 5.3.

It is important to remind the reader that RD analysis does not model timing, so our profiler's insights do not cover how multi-level designs impact end-to-end performance. In particular, our intent is not to compare the performance of different latency-*vs*-cost points in the design space (*e.g.*, L2s implemented in stacked DRAM as in Figure 21, eDRAM as in PS-Dir, or system DRAM as in WayPoint). This would require detailed architectural simulation. Instead, our goal is to shed light on how multi-level designs affect other metrics, like the access rate or miss rate of the L1 and L2 directory caches.

7.4.1. Methodology. To conduct our experiments, we modified the cache simulator from Sections 6.1 and 7.1 to model a multi-level directory similar to the one depicted in Figure 21. In the modified simulator, we assume the multicore CPU is attached to a stacked DRAM die. We use the exact same 3-level private cache hierarchy on the CPU die that was assumed in previous sections of this article whose parameters are specified in the top portion of Table VII. As shown in Table VII, the L1 and L2 caches have fixed capacities, but we vary the L3 cache size across four different capacities between 256 KB – 2 MB. In addition, we assume the DRAM die implements a large level-4 shared cache. The L4 shared cache is 1 GB in size and 8-way set associative.

Instead of a monolithic Cuckoo directory as was assumed in Sections 6.1 and 7.1, we model a multi-level directory cache consisting of two simple sparse directories [Gupta et al. 1990] (*i.e.*, without any fancy hashing). The L1 directory cache is 6-way set associative, implemented on chip along with the cores' private cache hierarchy. Its capacity varies with L3 cache sizing as specified in Table VII, but we always maintain an 18.75% coverage over the L3 cache size, whatever it may be. The L2 directory cache is an in-cache directory, integrated with the L4 shared cache. Notice, this ensures that there are never any directory-induced evictions. Because we assume an inclusive cache hierarchy, data blocks that reside in any private cache must also be resident in the shared L4 cache. Thus, they are guaranteed at least an L2 directory entry (if not an L1 directory entry). Full-map entries are used in both the L1 and L2 directory caches.

The directory caches are managed as follows. On a directory access, the L1 directory cache is checked first, and if the L1 misses, than the L2 directory cache is checked. On

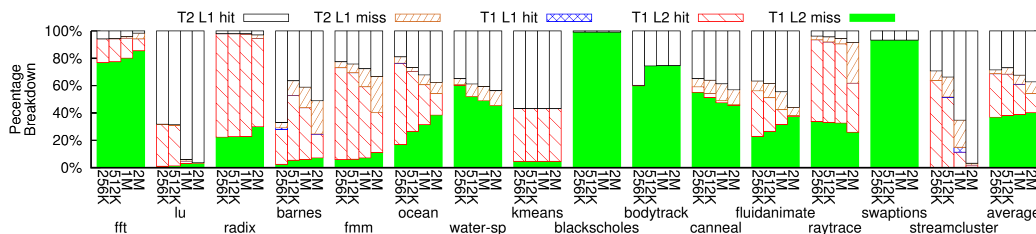


Fig. 22: Breakdown of T1 and T2 transactions in terms of hits and misses in the L1 or L2 of the multi-level directory cache. Breakdowns are reported for different private data cache sizes.

an L2 miss, a new directory entry is filled into the L1 directory cache. Due to in-cache integration of the L2 directory, such an L2 miss will always be a T1 transaction and will initiate a new lifetime in the multi-level directory cache. The filled directory entry will then receive zero or more T2 transactions during its time in the L1 directory cache (*i.e.*, low-latency L1 hits). Later, the entry will be evicted from the L1 and written to the L2 directory cache. On an L2 directory cache hit, the directory entry is promoted back into the L1. This can be from either a T1 or T2 transaction, depending on whether or not the associated data is still in the private data caches.

When the data associated with a directory entry is evicted from the private data cache, an attempt is made to notify the directory. In particular, if the associated directory entry is found in the L1 directory cache, the sharer list is updated to reflect the smaller sharing set. And, if the sharer list becomes empty, the directory entry is invalidated to make room for other entries in the L1 directory cache. If however, the directory entry is not found in the L1 directory cache, the L2 directory cache is notified only when evicting dirty blocks. Clean evictions do not notify the L2 in the interest of minimizing traffic to the DRAM die. So, the L2 directory cache may contain stale entries whose data blocks are actually in the invalid state but the directory entries are retained due to imprecise tracking.

7.4.2. Results. Figure 22 shows our multi-level directory cache results. The figure breaks down the directory's accesses into five categories: T2 transactions that hit in the L1 directory cache, T2 transactions that miss in the L1 directory cache (and thus hit in the L2 directory cache), T1 transactions that hit in either the L1 directory cache or the L2 directory cache,[6] and T1 transactions that miss in both directory caches. These categories are labeled "T2 L1 hit," "T2 L1 miss," "T1 L1 hit," "T1 L2 hit," and "T1 L2 miss," respectively. Groups of bars report the breakdowns across the four L3 cache sizes we simulated for each benchmark.

In Figure 22, we see that the great majority of T2 transactions hit in the L1 directory cache. As the "T2 L1 hit" and "T2 L1 miss" components show, the hit rate for T2 transactions is between 82–91% across different data cache sizes on average. Only 15% of the T2 transactions on average miss in the L1 directory cache and require going to the L2 directory cache. (In Table IX, the detailed per-benchmark T2 hit rates for the 256 KB and 1 MB bars from Figure 22 are reported in the two columns labeled "T2 h.r. Sim"). So, a small L1 directory cache can indeed satisfy most of the T2 transactions, demonstrating the same insight that was presented in Section 5.3.

[6]T1 transactions are destined to data blocks in the invalid state, so they normally miss in the directory cache. But directory entries do not receive eviction notifications while resident in the L2 directory cache. So, even when a data block transitions to invalid state, its directory entry may not reflect that. Such stale directory entries can linger, allowing some T1 transactions to hit in either the L1 or L2 directory cache.

Effectively handling T2 transactions is important for CPU performance. As mentioned in Sections 2 and 5.3, T2s are on-chip transactions whose directory lookups are on the critical path of the CPU's memory accesses. (They identify any on-chip sharers associated with a core's cache misses, and are needed for sharing-based transactions to continue). Fortunately, their high temporal reuse of directory entries as shown in Figure 22 implies that a multi-level directory will not significantly slow down these latency-sensitive directory accesses.

While T2 hit rates are high, the hit rates for T1 transactions in the L1 directory cache are near zero, as shown by the "T1 L1 hit" components in Figure 22. Moreover, T1 transactions dominate T2 transactions. So, when considering all directory accesses, the hit rate for the L1 directory cache is quite low, between 28–32% across different data cache sizes on average. (In Table IX, the detailed per-benchmark total hit rates for the 256 KB and 1 MB bars from Figure 22 are reported in the two columns labeled "Total h.r. Sim"). Again, this is an insight that was presented in Section 5.3 which we can now see demonstrated in Figure 22.

But the traffic coming out of the L1 directory cache consists mostly of T1 transactions since so few T2 transactions miss in the L1 directory cache. As discussed in Sections 2 and 5.3, T1s are off-chip transactions that are associated with slower memory accesses. For the system in Figure 21(b), they involve at the very least a shared L4 cache access, and in the worst case, a main memory access. So, any overhead incurred for checking the L2 directory cache is associated with an already expensive memory operation. Moreover, the overhead can usually be hidden. Granted, while it is not certain that an L1 directory cache miss is caused by a T1 transaction until after checking the L2 directory cache, the fact remains most L1 misses are due to T1s. So, one can *speculate* this all of the time, and always initiate an L4 shared cache access (and a DRAM access on an L4 miss) in parallel with the L2 directory cache access. When the speculation is correct, the L2 directory access will be overlapped with the data access; when the speculation is wrong, which is rare, the only harm done is a data block may be fetched needlessly. Hence, in a multi-level directory, one can say that while L1 directory cache misses are frequent, they consist mostly of T1 transactions that are highly tolerant of the more expensive L2 directory cache accesses.

Finally, not only do the simulation results in Figure 22 agree qualitatively with the profiler insights from Section 5.3, there is also some agreement quantitatively. To illustrate, we re-visit the access distribution results in Figure 10. This figure reports the number of directory entries exhibiting different access frequencies in the directory cache. In particular, the graphs in Figure 10 plot the directory entries' coverage along the Y-axis sorted from most frequently accessed directory entries to directory entries with only a single access. To better show the data's agreement with Figure 22, we add to Table IX a re-interpretation of these results in which we compute the directory accesses that a fixed number of the most frequently accessed directory entries would receive. In other words, for a fixed coverage value or horizontal line in the graphs of Figure 10, we determine the fraction of directory accesses that is captured by the directory entries below that horizontal line–*i.e.*, the hit rate similar to what was observed in the simulator.

For all of the profiler-based results in Table IX, we assume a fixed coverage value of 18.75% to match the L1 directory cache size used in our simulations. In the table, we report the hit rates for T2 transactions alone (columns labeled "T2 h.r. Prof") and the hit rates for all directory accesses (columns labeled "Total h.r. Prof") for the most frequently accessed directory entries from Figure 10 that fit within the 18.75% coverage. Results are shown for the same two data cache sizes, 256 KB and 1 MB, allowing a side-by-side comparison with the simulated hit rates already discussed.

Benchmark	T2 h.r. Sim		T2 h.r. Prof		Total h.r. Sim		Total h.r. Prof	
	256KB	1MB	256KB	1MB	256KB	1MB	256KB	1MB
fft	98.0	75.7	100.0	100.0	5.8	4.0	19.3	19.3
lu	99.4	99.0	100.0	100.0	68.1	94.1	69.3	98.4
radix	93.3	78.7	100.0	100.0	2.0	2.0	21.1	24.8
barnes	94.5	73.1	100.0	73.5	68.3	41.2	37.0	42.5
fmm	84.1	67.6	100.0	86.9	22.5	27.6	22.9	35.6
ocean	79.9	81.9	100.0	100.0	18.9	32.2	26.2	40.6
water-spatial	88.5	79.2	98.7	97.4	34.8	40.5	21.6	54.5
kmeans	99.8	99.9	100.0	100.0	56.8	56.9	76.1	73.7
blackscholes	100.0	100.0	100.0	100.0	0.8	0.8	20.4	19.5
bodytrack	100.0	100.0	100.0	100.0	39.7	25.3	100.0	100.0
canneal	85.2	75.6	100.0	93.2	34.9	38.7	30.0	46.0
fluidanimate	83.4	77.2	86.4	87.0	36.6	44.5	37.2	54.2
raytrace	58.5	57.7	100.0	100.0	3.8	5.5	24.9	27.1
swaptions	99.8	100.0	100.0	100.0	6.6	6.7	24.4	24.8
streamcluster	81.0	76.6	93.9	97.4	29.4	68.7	45.1	95.7
Average	89.7	82.8	98.6	95.7	28.5	32.3	38.4	50.5

Table IX: Percentage of T2 and total directory accesses that hit in a directory cache with 18.75% coverage for 256 KB and 1 MB private data caches, for the simulator results in Figure 22 (Sim) and the profile results in Figure 10 (Prof).

Table IX shows there are several cases for which the profiler and simulator hit rates differ. Significant differences occur in raytrace for the T2 hit rate, and bodytrack for the total hit rate. There are also many other cases with non-trivial differences: about 20-25% discrepancy in T2 hit rates for fft, radix, and streamcluster (1 MB) and ocean (both 256 KB and 1 MB); and about 20% discrepancy in total hit rates for fft, radix, kmeans, blackscholes, raytrace, and swaptions. These errors are due to the fact that the profiler results represent the best-case scenario in which an oracle keeps the most frequently accessed directory entries in the L1 directory cache. In contrast, the simulations employ actual replacement policies that may occasionally evict important directory entries. (Notice, the simulator hit rates are almost universally lower than the profiler hit rates, which is consistent).

But overall, there is agreement between the profiler and simulator results. Like the simulator, the profiler shows very good hit rates for T2 transactions alone (on average, 98.6% – 95.7% as compared to 89.7% – 82.8% for the simulator). At the same time, it shows poor hit rates for all directory accesses (on average, 38.4% – 50.5% as compared to 28.5% – 32.3% for the simulator).

7.5. Directory Access Count Discussion

The case studies in Sections 7.1–7.4 have focused on improving the directory's capacity scaling, a very important design consideration for directory caches. But another important consideration is the directory's access rate. Regardless of which directory technique a designer adopts, the rate of the directory accesses will have far-reaching implications for the directory's performance.

Fundamentally, the down-stream traffic from a cache reduces as its capacity goes up. This is well understood for data caches, especially in uniprocessors. Our analyses from Section 5.1 quantify this bandwidth reduction for parallel caches (*i.e.*, the part that is incident on the directory) and how to break down its components. This can help architects make design tradeoffs in directory caches as CPUs scale.

For example, our main observation–that the total number of directory accesses drops with CPU scaling, especially cache size scaling–implies that directory cache accesses

will make up a smaller fraction of overall execution time as CPUs scale. Among the directory designs discussed in this article, many propose techniques for reducing directory size that also increase the cost of directory lookups. In particular, techniques like Cuckoo and also tagless directories [Zebchuk et al. 2009] employ complex hash functions to reduce over-provisioning. And, techniques like SCD as well as other hierarchical directories [Guo et al. 2010] may require multiple directory accesses to lookup entries that encode wide sharing (*i.e.*, to access root entries versus leaf entries). Our results show that trading off increased access latency to achieve smaller directories is a good idea as CPUs scale.

Another important observation in Section 5.1 is that the mix of T1 *vs.* T2 transactions varies significantly with CPU scaling. Recently, researchers have explored increasing the effective associativity of directory caches by employing iterative reinsertion techniques. For example, both Cuckoo and SCD employ such techniques. These techniques dramatically reduce conflicts at the expense of more costly insertions. While insertions (T1s) constitute the vast majority of directory accesses in small data caches, they become much less frequent in large data caches due to increased sharing and T2-based reuse. So, our results show that trading off more complex insertion algorithms to mitigate conflicts is also a good idea as CPUs scale.

8. RELATED WORK

This article is based on our own previous work which was the first to extend multicore reuse distance analysis for handling coherence directories [Zhao and Yeung 2015]. Compared to that earlier paper, this article provides a more extensive set of profile-based results and insights as well as a more complete set of validation experiments. In addition, we present new case studies for SCD and DGD directories in Section 7. We also present a new multi-level directory technique, and feature it in a third new case study.

There has also been significant prior research on multicore reuse distance analysis. Our work exploits PRD profiling [Schuff et al. 2009; Schuff et al. 2010; Wu et al. 2013; Wu and Yeung 2013] which analyzes private caches. There has also been work on analyzing shared caches using concurrent research distance (CRD) profiling [Ding and Chilimbi 2009; Jiang et al. 2010; Wu and Yeung 2011; Wu et al. 2013; Wu and Yeung 2013]. But all of these previous multicore RD efforts have focused on the data cache hierarchy. This article, along with our previous paper mentioned above, are the first to apply reuse distance for reasoning about directory caches.

Researchers have also developed analytical models for directory caches. For example, the Cuckoo work [Ferdman et al. 2011] used a model to estimate the directory's size and energy consumption as core count and data cache size are simultaneously scaled. In addition, SCD [Sanchez and Kozyrakis 2012] presented a model for estimating over-provisioning in the directory cache. Compared to RD analysis, analytical models are much faster. They are also general in that they are not tied to specific applications. However, because RD analysis employs profiling, it can detect application-specific behaviors and account for them.

Finally, there is a large body of research on directory caches (see Section 1), and some have studied the effects of multicore scaling on their techniques using simulation. Most only simulate a single core count and cache size [Cuesta et al. 2011; Cuesta et al. 2013; Guo et al. 2010; Gupta et al. 1990; Sanchez and Kozyrakis 2012; Valls et al. 2012; Zebchuk et al. 2009; Zebchuk et al. 2013]. Cuckoo and SCT [Alisafaee 2012] simulated 2 different private cache sizes at a single core count. WayPoint [Kelm et al. 2010] simulated 6 different core counts, but only for a single cache size. In comparison, our work studies a much larger cross product of cache sizes and core counts. More importantly, we characterize scaling's impact at the source–*i.e.*, the directory's access

stream. Hence, our insights are applicable to many techniques, not just a single specific directory design.

9. CONCLUSION

This article applies multicore reuse distance analysis to analyze directory caches, and uses it to extract insights on CPU scaling. A key contribution we make is the notion of relative reuse distance between sharers. This enables quantifying how sharing evolves with CPU scaling, which yields the directory's access patterns and contents at different scaling points.

We build a profiler that implements our RD analyses for directory caches, and use it to study several parallel benchmarks. We find the number of directory accesses drops by 3.3x on average when scaling a CPU's private data caches from 16 KB to 1 MB. This drop occurs despite the fact that data cache scaling captures more sharing on chip, resulting in more sharing-based directory accesses. In addition, we also find the increased on-chip sharing reduces the number of unique directory entries needed to track all sharers, allowing coverage to reduce by 43.3% across data cache size scaling. In contrast, core count scaling increases the number of directory accesses by only 38.5% and decreases coverage by only 2.6% on average when scaling from 16 to 256 cores. Furthermore, we find certain directory entries exhibit high temporal reuse, allowing 23.0% of the entries to receive 42.7% of all directory accesses and 83.1% of T2-based accesses on average for a 64-core CPU with 1 MB private data caches. We also validate these profile-based results, showing they are within 2-10% of cache simulation on average across different validation experiments.

Lastly, we conduct several case studies to illustrate our scaling insights on existing directory techniques. First, we show as data cache size scales, a Cuckoo directory can be reduced down to only 37.5–87.5% coverage for most benchmarks, tracking our profiler's predictions. Second, we show our profiler's coverage results provide a lower bound on the number of directory entries in an SCD directory. We find many benchmarks achieve this lower bound. But, some benchmarks allocate noticeably more directory entries than this lower bound. Third, we show our profiler's coverage for multi-sharer directory entries provides a lower bound on the amount of coalescing that a DGD directory can achieve. At small data cache sizes, most cache blocks are private, so there is ample opportunity for coalescing. But at large cache sizes, many private cache blocks become shared, so the opportunity for coalescing diminishes and DGD is unable to achieve the lower bound for many benchmarks. And fourth, we show that a majority of T2-based directory accesses hit in the L1 directory cache of a multi-level directory cache, which is in agreement with our profiler's access distribution results.

In the future, we hope to extend our research. One direction for future work is to develop new techniques based on our profiling insights. In particular, Section 7.1 demonstrated potential benefits for dynamically resizing directory caches to exploit application-dependent drops in coverage. Such techniques could reduce the directory's power consumption. Another direction for future work is to study a broader range of applications and/or problem sizes. As shown in Section 5, our benchmarks are dominated by accesses to truly private data, with modest amounts of sharing and a lack of widely shared data. It would be interesting to see how our results would change for other applications exhibiting greater degrees of sharing. At the same time, our current work only considers fixed problem sizes. For a given range of cache capacities, problem size scaling would probably further decrease the amount of sharing we currently observe in our results. This could reduce the magnitude of the drops in coverage reported in Sections 5.2 and 7.1, and hence, diminish the opportunity for resizing directory caches as data caches are scaled.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments, and Meng-Ju Wu for advice and discussions.

REFERENCES

ACACIO, M. E., GONZALEZ, J., GARCIA, J. M., AND DUATO, J. 2001. A New Scalable Directory Architecture for Large-Scale Multiprocessors. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*. Washington, D.C.

AGARWAL, A., BAO, L., BROWN, J., EDWARDS, B., MATTINA, M., MIAO, C.-C., RAMEY, C., AND WENTZLAFF, D. 2007. Tile Processor: Embedded Multicore for Networking and Multimedia. In *Proceedings of the Symposium on High Performance Chips*.

AGARWAL, A., SIMONI, R., HENNESSY, J., AND HOROWITZ, M. 1988. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*. Los Alamitos, CA.

ALBONESI, D. H. 1999. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*. 248–259.

ALISAFAEE, M. 2012. Spatiotemporal Coherence Tracking. In *Proceedings of the 45th Annual International Symposium on Microarchitecture*.

BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. Vancouver, Canada, 282–293.

BERG, E., ZEFFER, H., AND HAGERSTEN, E. 2006. A statistical multiprocessor cache model. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*. 89–99.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

CHAIKEN, D., KUBIATOWICZ, J., AND AGARWAL, A. 1991. 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.

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

CHOI, J. H. AND PARK, K. H. 1999. 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.

CUESTA, B., ROS, A., GOMEZ, M. E., ROBLES, A., AND DUATO, J. 2013. Increasing the effectiveness of directory caches by avoiding the tracking of noncoherent memory blocks. *IEEE Transactions on Computers 62,* 3, 482–495.

CUESTA, B. A., ROS, A., GÓMEZ, M. E., ROBLES, A., AND DUATO, J. F. 2011. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th annual international symposium on Computer architecture*. ACM, New York, NY, USA, 93–104.

DING, C. AND CHILIMBI, T. 2009. A Composable Model for Analyzing Locality of Multi-threaded Programs. Technical Report MSR-TR-2009-107, Microsoft Research.

EKLOV, D., BLACK-SCHAFFER, D., AND HAGERSTEN, E. 2011. Fast modeling of shared caches in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. HiPEAC '11. ACM, New York, NY, USA, 147–157.

FERDMAN, M., LOTFI-KAMRAN, P., BALET, K., AND FALSAFI, B. 2011. Cuckoo directory: A scalable directory for many-core systems. In *17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 169–180.

GUO, S.-L., WANG, H.-X., XUE, Y.-B., LI, C.-M., AND WANG, D.-S. 2010. Hierarchical Cache Directory for CMP. *Journal of Computer Science and Technology 25,* 2, 246–256.

GUPTA, A., DIETRICH WEBER, W., AND MOWRY, T. 1990. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *In International Conference on Parallel Processing*. 312–321.

HSU, L., IYER, R., MAKINENI, S., REINHARDT, S., AND NEWELL, D. 2005. Exploring the Cache Design Space for Large Scale CMPs. *ACM SIGARCH Computer Architecture News 33*.

INTEL. 2014. Intel Xeon Phi Product Family.

JIANG, Y., ZHANG, E. Z., TIAN, K., AND SHEN, X. 2010. Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors? In *Proceeding of Compiler Construction*.

KELM, J. H., JOHNSON, M. R., LUMETTTA, S. S., AND PATEL, S. J. 2010. Waypoint: Scaling coherence to thousand-core architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. ACM, New York, NY, USA, 99–110.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

MADAN, N., ZHAO, L., NAVEEN MURALIMANOHAR, UDIPI, A., BALASUBRAMONIAN, R., IYER, R., MAKINENI, S., AND NEWELL, D. 2009. Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy. In *Proceedings of the International Symposium on High Performance Computer Architecture*.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design*. Rapallo, Italy.

MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9,* 2, 78–117.

MCCURDY, C. AND FISCHER, C. 2005. Using pin as a memory reference generator for multiprocessor simulation. *ACM SIGARCH Computer Architecture News 33*.

NARAYANAN, R., OZISIKYILMAZ, B., ZAMBRENO, J., MEMIK, G., AND CHOUDHARY, A. 2006. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proceedings of the International Symposium on Workload Characterization*.

POWELL, M., YANG, S.-H., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. N. 2000. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics & Design*. 90–95.

SANCHEZ, D. AND KOZYRAKIS, C. 2012. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*.

SCHUFF, D. L., KULKARNI, M., AND PAI, V. S. 2010. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*.

SCHUFF, D. L., PARSONS, B. S., AND PAI, J. S. 2009. Multicore-Aware Reuse Distance Analysis. Technical Report TR-ECE-09-08, Purdue University.

VALLS, J. J., ROS, A., SAHUQUILLO, J., GÓMEZ, M. E., AND DUATO, J. 2012. Ps-dir: a scalable two-level directory cache. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. PACT '12. ACM, New York, NY, USA, 451–452.

WALLACH, D. A. 1993. PHD: A Hierarchical Cache Coherent Protocol (Master's Thesis).

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*.

WU, M.-J. AND YEUNG, D. 2011. 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.

WU, M.-J. AND YEUNG, D. 2013. Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs. *ACM Transactions on Computer Systems 31,* 1.

WU, M.-J., ZHAO, M., AND YEUNG, D. 2013. Studying Multicore Processor Scaling via Reuse Distance Analysis. In *Proceeding of the International Symposium on Computer Architecture*. Tel-Aviv, Israel.

YANG, S.-H., FALSAFI, B., POWELL, M. D., AND VIJAYKUMAR, T. N. 2002. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. HPCA '02. IEEE Computer Society, Washington, DC, USA, 151–161.

ZEBCHUK, J., FALSAFI, B., AND MOSHOVOS, A. 2013. Multi-Grain Coherence Directories. In *Proceedings of the 46th Annual International Symposium on Microarchitecture*. Davis, CA.

ZEBCHUK, J., SRINIVASAN, V., QURESHI, M. K., AND MOSHOVOS, A. 2009. A Tagless Coherence Directory. In *Proceedings of the 42nd International Symposium on Microarchitecture*. New York, NY.

ZHAO, L., IYER, R., MAKINENI, S., MOSES, J., ILLIKKAL, R., AND NEWELL, D. 2007. Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*.

ZHAO, M. AND YEUNG, D. 2015. Studying the Impact of Multicore Processor Scaling on Directory Techniques via Reuse Distance Analysis. In *Proceeding of the 21st International Symposium on High Performance Computer Architecture*. San Francisco Bay Area, CA.