Appears in *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA-XXI). San Francisco Bay Area, CA. February 2015.*

Studying the Impact of Multicore Processor Scaling on Directory Techniques via Reuse Distance Analysis

Minshu Zhao and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park
{mszhao,yeung}@umd.edu

***Abstract*—Researchers have proposed numerous directory techniques to address multicore scalability whose behavior depends on the CPU's particular configuration, *e.g.* core count and cache size. As CPUs continue to scale, it is essential to explore the directory's architecture dependences. However, this is challenging using detailed simulation given the large number of CPU configurations that are possible.**

This paper 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. We also implement our framework in a profiler, and apply it to gain insights into multicore scaling's impact on the directory.

Our profiling results show that directory accesses reduce by 3.5x across data cache size scaling, suggesting techniques that tradeoff access latency for reduced capacity or conflicts become increasingly effective as cache size scales. We also show the portion of on-chip memory devoted to the directory cache can be reduced by 53.3% across data cache size scaling, thus lowering the over-provisioning needed at large cache sizes. Finally, we validate our RD-based directory analyses, and find they are within 13% of cache simulations in terms of access count, on average.

I. 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 [18] and Tilera's Tile processors [1], already implement 10s of cores on a single die. In the future, processors with 100s of cores, *i.e.* large-scale chip multiprocessors [17], [36], will be possible.

To enable scaling, architects have investigated directory-based cache coherence. An important factor in the scalability of these protocols is the design of their coherence directories. Duplicate tag directories [4] are impractical because they require high associativity as CPUs scale. In contrast, sparse directories [16] 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 superlinearly. Worse yet, sparse directories incur conflicts, 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 tracked imprecisely using limited pointers [2], [7], [8], [9], coarse vectors [16], or tagless arrays [35]. Also, the directory can be implemented hierarchically [15], [29], resulting in logarithmic sharer list growth. More recently, SCD [25] combines limited pointers with hierarchical lists to compactly encode both narrow and wide sharing patterns. In addition to these techniques, researchers have also tried to *reduce conflicts*. For example, Cuckoo [14] and SCD [25] use multiple hash functions and iterative re-insertion to increase associativity.

Another important approach is *exploiting private data*. Cuesta's work [11], [10] detects pages that are accessed by a single core, and omits tracking their cache blocks. PS-Dir [28] devotes a separate directory to track private data using minimally-sized sharer lists. And, SCT [3] and MGD [34] recognize private data tend to occur in large contiguous regions. Hence, they coalesce consecutive privately accessed cache blocks, 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 [28] provides a fast directory in SRAM for frequently accessed directory entries, and a slow directory in denser eDRAM for infrequently accessed entries. Also, WayPoint [20] evicts infrequently accessed entries that do not fit in the on-chip directory to off-chip DRAM.

The effectiveness of these 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.*, the CPU's configuration. For instance, varying core count will 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–in particular, scaling capacity–will change the directory's behavior. Specifically, *it can alter the perceived*

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.

sharing patterns. While sharing is inherently an application 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 application and architecture dependences. 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-dependent 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 [3], [14], [20]. 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 locality analysis [5], [12], [19], [27], [26], [13], [32], [33]. These techniques acquire *reuse distance (RD) profiles*. A program's RD profile is its memory reuse distance histogram, capturing the 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 [27], [26], [32] uses per-thread coherent LRU stacks to model the interactions that occur across private data caches. The key is such profiles are architecture independent across cache size scaling, and highly predictable across core count scaling [32]. So, a few profiles can analyze caching behavior across a large number of CPU configurations without having to simulate them.[1]

In this paper, we apply multicore RD analysis to study coherence directories. Our goal is to enable for coherence 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 to allow analysis of the directory's access patterns and contents. A key notion we develop is *relative reuse distance between sharers*, which quantifies sharing in a capacity-sensitive fashion. Due to the cache-size independence of PRD, we can perform analyses at every possible private data cache size from a single profile. Also, using existing insights on PRD profiles [33], we can analyze the directory's behavior across core count scaling as well.

We implement our analyses in a PIN-based profiler [21], and use it to study directory behavior. Our profiling results show a 3.5x drop in directory accesses occurs when data cache size scales from 16KB to 1MB, despite an increase in sharing-based directory accesses. We also find elevated sharing reduces the number of active directory entries needed to track all sharers, allowing the portion of on-chip memory devoted to the directory to decrease by 53.3% across data cache size scaling. This trend is accompanied by a significant reduction in the number of directory entries for private data relative to shared data. In addition to cache size scaling, we find core count scaling at a fixed capacity only increases the directory's accesses by 38% despite a 16x increase in core count and decreases the directory size by 2.6% despite a 4x increase in core count.

To validate our profiling results, we compare them against cache simulations. Our validation experiments show the profiled directory access counts are within 9.3% of simulation across cache size scaling on average, and 13% across core count scaling. Moreover, the profiled directory sizes are within 3.0% of simulation across cache size scaling on average, and 3.7% across core count scaling. Finally, we discuss the implications of our profiling results for existing directory techniques. One implication is that reducing directory size at the expense of more costly lookups is a desirable tradeoff as CPUs scale. Another implication is the fraction of on-chip memory needed for the directory varies significantly with scaling, especially cache size scaling. We show for most benchmarks, a Cuckoo directory only needs to provide entries for 37.5–87.5% of cache blocks in the private data caches.

The rest of this paper is organized as follows. Section II discusses the directory accesses we analyze. Then, Section III presents our analysis framework, and Section IV implements it. Next, Section V reports our profiling results while Section VI validates them and discusses their implications for directory techniques. Finally, Section VII covers related work and Section VIII concludes the paper.

II. Directory Accesses

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 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 transac-

[1]PRD is sensitive to reference interleaving, so strictly speaking, it is architecture dependent. But research has shown that changes in interleaving are benign for programs with symmetric threads and loop-level parallelism [19], [32]. So, PRD is accurate for these programs.

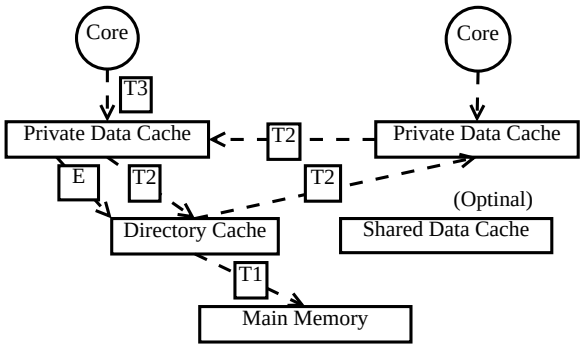


Fig. 1. Directory accesses in a multicore cache hierarchy.

tions 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 their modifications to the directory, are determined by the private data caches. Hence, they are architecture dependent. Specifically, scaling the number of private caches (*i.e.*, cores) and their capacity 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.

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 III-C will discuss the impact of this on our analyses.

III. Analysis Framework

This section presents our RD-based framework for analyzing multicore scaling's impact on directory caches. Section III-A reviews multicore RD techniques. Then, Sections III-B and III-C develop new analyses to identify the directory accesses and modifications discussed in Section II.

A. 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

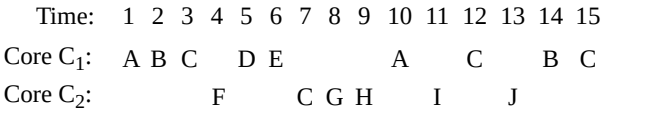


Fig. 2. Two interleaved memory reference streams.

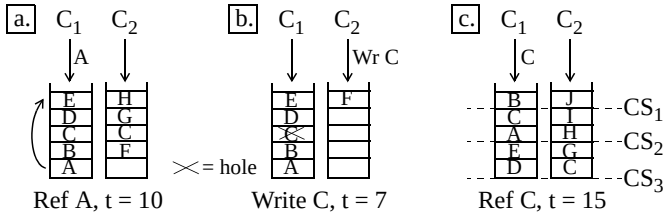


Fig. 3. LRU stacks illustrating (a) intra-thread reuse and replication, (b) invalidation, and (c) PRD$_{remote}$.

referenced since the last reference to the same data block.[2] 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 for capacity CS. (This calculation accurately predicts misses in set associative caches too, as long as capacity misses dominate conflict misses). One of the major benefits of RD profiles is that 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 [27], [26], [32], [33] 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 the memory references from two cores, C_1 and C_2, performed on data blocks A–J, and Figure 3 shows the corresponding LRU stacks at different times. In Figure 3(a), we see C_1's re-reference of A at $t = 10$, assuming all references in Figure 2 are reads. Block A is found below blocks B–E in C_1's LRU stack, so we say its PRD = 4. A cache of size 5 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.

In addition to intra-thread reuse, PRD profiling also captures inter-thread interactions, such as sharing. For read sharing, PRD captures the resulting replication effects across LRU stacks. In Figure 2, both C_1 and C_2 access data block C. Assuming these are both reads, Figure 3(a) shows the C block is replicated in the cores' stacks. Such shared replicas increase the capacity pressure within the affected stacks.

PRD also captures write sharing effects by maintaining coherence between LRU stacks. For example, suppose C_2's reference to C at $t = 7$ is a write instead of a read. Then, invalidation would occur in C_1's stack, as shown in

[2]Reuse distance has also been referred to as "stack distance" because RD calculations are performed on memory reference stacks [22].

Figure 3(b). To prevent promotion of blocks further down the LRU stack, invalidated blocks leave behind "holes" [27]. 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 re-references A at $t = 10$, E and D in Figure 3(b) will be pushed down and the hole will move to depth 4 (A's old position), preserving the stack depth of B. After the invalidation, C_1's re-reference of C at $t = 12$ will miss regardless of the cache capacity–*i.e.* a coherence miss–so we say its PRD = ∞.

Besides analyzing cache misses under capacity scaling, PRD can also analyze core count scaling effects. As the number of cores increases, PRD profiles often change systematically: they shift to larger RD values in a shape-preserving fashion [32], [33]. 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 minimal for very large caches. This is because memory references with small RD values tend to access private data, whereas shared references tend to exhibit large RD values. Thus, core count scaling increases cache pressure more at small cache capacities. (Details on these effects can be found in prior work [32]).

In the remainder of this section, we extend 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 V, 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.

B. Directory Access Analysis

Because PRD profiling can predict private data cache misses, it can identify cache miss-induced directory accesses. In particular, given a reference's PRD and access mode (read or write), we can predict whether a cache miss and directory access will occur at cache size CS, and if so, its type.

Consider the examples from Section III-A. In Figure 3(a), if C_1's private cache is sufficiently large to capture the reuse on block A (PRD $< CS$), then the reference hits–a T3 transaction–and no directory access occurs. Otherwise (PRD $\geq CS$), the reference misses and generates a directory access. Given there are no other copies of A on-chip, this is a T1 transaction that initiates a new directory entry lifetime. In Figure 3(b), if C_2's reference to block C is a write, then the references at $t = 7$ and 12 would both miss and generate directory accesses. (Like Figure 3(a), these also depend on the cache size CS, which we will address next). Since these are due to inter-thread communication, they are T2 transactions that reuse the directory entry inserted at $t = 3$.

One issue PRD profiling does not address is sharing's dependence on cache size. Granted, sharing is an application-level property. But even if threads share data, whether or not the sharing manifests on-chip depends on cache size. So, the number of sharing-based T2 transactions is tied to temporal locality–*i.e.*, to the *relative reuse distance between sharers*.

Figure 3(c) illustrates this by showing C_1's reuse of block C at $t = 15$. Both cores have the block in their LRU stacks, but C_1 has referenced it more recently than C_2. 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 neither copy is on-chip, so C_1's reference misses and generates a T1 directory access. If the cache size is CS_2, then only C_1's copy is on-chip. We say block C is "temporally private" [3]–*i.e.* it is private within the limited time window captured by CS_2. So, C_1's reference is a hit regardless of access mode (a T3) with no directory access. Lastly, if the cache size is CS_3, then both copies are on-chip. While a read would again be a T3 transaction, a write would cause a T2 directory access.

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 PRD$_{remote}$ = ∞, 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, PRD$_{remote} < CS$ would mean the sharing is captured; otherwise (PRD$_{remote} \geq CS$), the memory reference is temporally private.

1) Access Mode, PRD, PRD$_{remote}$ Characterization.: Table I lists all data cache transactions that can occur by permuting the access mode (read or write) and the different PRD/PRD$_{remote}$ outcomes ($< CS$, $\geq CS$, and ∞) discussed above. In total, there are 18 different cache transactions. Table I reports all of them in terms of the T1–T3 categories.

The first eight transactions in Table I form the T1 category. All of these are misses in the local private cache and in all remote private caches (PRD and PRD$_{remote} \geq CS$); therefore, there is no sharing captured on-chip. Transactions 1 and 2 are cold misses. While transactions 3 and 4 can be a local cold miss, in most cases they are coherence misses, as explained in Section III-A, which are re-references after write invalidations. Transactions 5 and 6 represent the case where the data is truly private and resides in the local cache–these correspond to Figure 3(a) assuming $CS < 5$. And, transactions 7 and 8 represent temporally private data–these correspond to Figure 3(c) assuming $CS = CS_1$.

The next five transactions in Table I form the T2 category. All of these exhibit sharing that is captured on-chip (PRD$_{remote} < CS$) and some remote action is required–either invalidation or forwarding of the requested block. Transactions 9 and 10 represent a read miss in the local private cache, but the data can be forwarded by a remote private cache. For example, transaction 9 corresponds to Figure 3(b) assuming the access is a read and $PRD_{remote} < CS$. Transactions 11, 12, and 13 represent a write to a shared block on chip, which causes invalidation. For example, transaction 13 corresponds to Figure 3(c) assuming the access is a write and $CS = CS_3$. Moreover, similar to transaction 3 and 4, transaction 9 and 11 can be coherence misses too.

The last five transactions form the T3 category. All of these are hits in the local private cache (PRD $< CS$) and do not require remote actions. Transactions 14 and 15 correspond to Figure 3(a) assuming $CS \geq 5$; transactions 16 and 17 correspond to Figure 3(c) assuming $CS = CS_2$; and transaction 18

TABLE I. THE 18 POSSIBLE DATA CACHE TRANSACTIONS.

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

corresponds to Figure 3(c) assuming $CS = CS_3$.

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(c), block B is evicted from a cache of size CS_1 after the reference to C pushes it down the stack. Suppose C_1 references block E instead of C in Figure 3(c). In that case, not only would B be evicted from a cache of size CS_1, but A would also be pushed down and evicted from a cache of size CS_2. As mentioned in Section II, whether or not a particular eviction notifies the directory depends on the coherence protocol. The next section will address this issue.

C. Directory Contents Analysis

The directory cache contents can be tracked by the same analyses from Section III-B. Initially, the directory cache is empty. As explained in Section II, 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 decreases by one.

If all data cache evictions notify the directory (a common assumption made in recent techniques [3], [14], [25], [34]), this analysis exactly tracks the directory's contents. However, if some evictions are silent–*e.g.*, for shared blocks–then our analysis is imprecise. We can still identify the notifications. But the silent evictions 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.

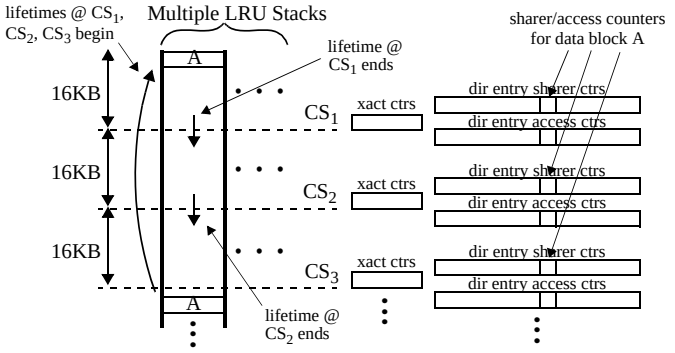


Fig. 4. Counters implemented in the PIN profiler.

IV. DIRECTORY CACHE PROFILER

We implemented RD-based directory cache profiling within the Intel PIN tool [21]. We modified PIN to maintain coherent private LRU stacks and perform PRD profiling, as discussed in Section III-A. (We assume 64-byte blocks in all LRU stacks). For every memory reference, our profiler consults the LRU stacks to compute PRD and PRD_{remote}, using Table I to determine the data cache transaction and directory access type.

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 and stop at the application's maximum PRD. For each CS value, we maintain 19 counters, one per transaction in Table I plus one for evictions, and increment the corresponding counter. Figure 4 illustrates the per-transaction counters, labeled "xact ctrs," at each profiled private cache size, labeled "CS_i."

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.

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, we increment the corresponding access counter.

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

TABLE II. PARALLEL BENCHMARKS USED IN THE EVALUATIONS. INSTRUCTION COUNTS, LABELED "INST," ARE REPORTED IN BILLIONS.

Benchmark	Suite	Problem Size	Inst
fft (kernel)	SPLASH2	2^{22} elements	2.46
lu (kernel)	SPLASH2	2048^2 elements	25.1
radix (kernel)	SPLASH2	2^{24} keys	3.15
barnes	SPLASH2	2^{19} particles	19.3
fmm	SPLASH2	2^{19} particles	16.5
ocean	SPLASH2	1026^2 grid	1.72
water	SPLASH2	40^3 molecules	1.86
kmeans	MineBench	2^{22} objects, 18 features	10.7
blackscholes	PARSEC	2^{22} options	3.94
bodytrack	PARSEC	B_261,16k particles	13.9
canneal	PARSEC	2500000.net	0.12
fluidanimate	PARSEC	in_500k.fluid	4.30
raytrace	PARSEC	1920x1080 pixels	4.39
swaptions	PARSEC	2^{18} swaptions	26.7
streamcluster	PARSEC	2^{18} data points	5.14

sharer counter belongs. And, the corresponding access counter reflects the number of accesses the directory entry received 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 shows how a reference to block A initiates directory entry lifetimes at capacities CS_1, CS_2, and CS_3, and how the first two lifetimes terminate as the block is pushed below capacities CS_1 and CS_2.

Finally, our PIN profiler follows McCurdy's method [23] which performs functional execution only, context switching threads after every memory reference. This interleaves threads' memory references uniformly in time. Studies have shown that for parallel programs with symmetric threads, this approach yields profiles that accurately reflect locality on real CPUs [19], [32], especially for PRD profiles.

V. 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 their suites: SPLASH2 [30], MineBench [24], and PARSEC [6]. The last two columns report the problem sizes and instruction counts (in billions). For the kernels–fft, lu, and radix–we profiled the entire benchmark. For other benchmarks, we ran the first parallel iteration to warm up the PRD stacks, and then profiled the second parallel iteration. We first study how scaling affects the directory's access stream and contents. Then, we study the distribution of accesses across the directory to show temporal reuse of directory entries. In the first two studies, we address cache size scaling followed by core count scaling. In the last study, we consider cache size scaling alone to show the main effects.

A. Study 1: Directory Access Results

Table III shows the impact of scaling private data cache size on cache miss-induced directory accesses (*i.e.*, T1 and T2 transactions) as reported by the "xact ctrs" in our profiler. In particular, columns 2–4 of Table III report the total number of cache miss-induced directory accesses per 1000 instructions, or "APKI," incurred by a 64-core CPU at 3 data cache sizes.

TABLE III. CACHE-MISS APKI AT 3 PRIVATE CACHE SIZES, INTRINSIC APKI, AND APKI FOR 16- AND 256-CORE CPUS.

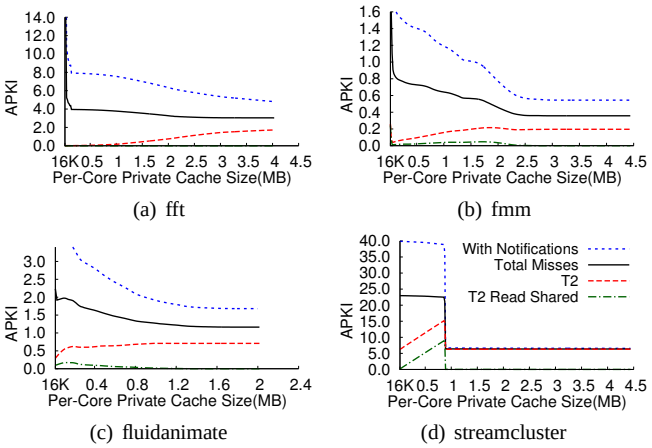
Benchmark	Cache Miss APKI			T2	APKI	
	16KB	256KB	1MB	∞	16c	256c
fft	16.0	3.9	3.8	1.7	3.7	3.9
lu	2.0	1.9	0.7	0.7	0.2	1.1
radix	16.7	5.7	5.7	2.3	5.6	6.1
barnes	19.1	0.9	0.8	0.6	0.6	0.9
fmm	2.0	0.8	0.6	0.2	0.6	0.8
ocean	32.0	15.9	7.1	2.0	6.0	10.1
water	2.4	1.5	0.6	0.2	0.5	0.8
kmeans	1.1	1.1	1.1	0.6	1.1	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	0.1	0.5	0.6
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	2.1	1.5	0.5	1.3	1.8

Table III shows directory cache accesses are highly sensitive to data cache size: *they drop rapidly as capacity increases*. For small 16KB private caches, about half the benchmarks in Table III exhibit a directory APKI exceeding 11 (reaching 32 in one case). But for 1MB caches, all benchmarks except for canneal exhibit a directory APKI of only 7.1 or less, with half under 1 APKI. Across all benchmarks, the average directory APKI drops from 5.3 at 16KB to 1.5 at 1MB, a factor of 3.5x.

Ostensibly, this drop is due to the reduction in cache misses that occurs when scaling cache sizes. But the reason is actually more nuanced, reflecting on how cache size scaling affects on-chip sharing. To illustrate, Figure 5 shows the complete behavior from our profiles, breaking down the directory's accesses as data cache size varies. The solid lines, labeled "Total Misses," plot APKI for all (T1 + T2) accesses; the dashed lines, labeled "T2," plot APKI for T2 accesses only; and the dash-dotted lines, labeled "T2 Read Shared," plot APKI for T2 accesses associated with read sharing (*i.e.*, transaction 10 in Table I). To save space, 4 representative benchmarks are shown.

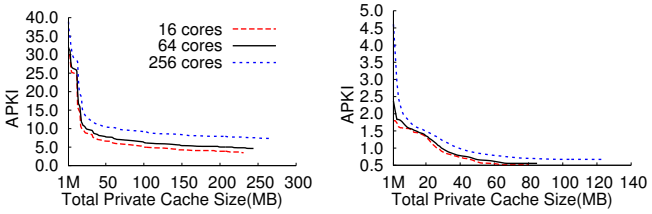
At small cache sizes, Figure 5 shows the directory cache accesses are dominated by T1 transactions (the gap between "Total Misses" and "T2"). The lack of T2s here makes sense since the data caches are too small to capture many shared accesses occurring between threads. So, the majority of references are to private data, regardless of whether they are truly or temporally 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, temporally private data blocks begin manifesting their sharing patterns on-chip, increasing the number of T2 transactions.

While T2 transactions generally go up with capacity, 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, the directory accesses are eliminated–*i.e.*, the read-sharing working set fits in cache. In contrast, write sharing leads to coherence-related T2 transactions. These also increase with capacity scaling, but they cannot be eliminated by capturing all sharers on-chip. This causes the gap between the "T2" and "T2 Read Shared"



(a) fft (b) fmm (c) fluidanimate (d) streamcluster

Fig. 5. Breakdown of directory APKI *v.s.* private cache size for 64-core CPUs.



(a) ocean (b) water

Fig. 6. Total directory APKI for 16-, 64-, and 256-core CPUs.

curves in Figure 5 to increase monotonically.

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 accesses. On average, they only reach 0.5 APKI. *Hence, while scaling 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, cache size scaling reduces 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. (Note, the X-axes in Figure 6 plot *total* cache size, instead of *per-core* cache size in Figure 5, to facilitate comparisons across core count).

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 III-A), which makes sense since directory accesses are derived from private cache misses that PRD profiles capture. Overall, the 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, 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. In most benchmarks, notifications 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. 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 directory accesses.

B. *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, as tracked by the number of sharer counters in our profiler. In particular, the solid lines, labeled "Total," plot the ratio of total live directory entries to total private cache blocks–a metric called *coverage* [25]–as data cache size varies. (Coverage at each cache size is time-averaged across the entire profiling run). All results are for 64-core CPUs.

As Figure 7 shows, *coverage–and hence, the portion of on-chip memory devoted to the directory–decreases significantly with cache size scaling*. In particular, coverage starts near 100% in most cases, but then drops to about 50% as cache size increases for many benchmarks. The effect is extreme in lu, bodytrack, and streamcluster, where coverage drops below 20%. On average, the coverage at each benchmark's maximum PRD is only 46.7%–*i.e.*, 53.3% of the directory does not contain active directory entries.

The drop in coverage is due to increased sharing that occurs at larger cache sizes. As discussed in Section V-A, 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 solid and $\geq$ 2 lines breaks down the coverage for single-sharer entries). For 64KB private caches, only 9.3% of directory entries are multi-sharer entries (over 90% are private entries) averaged across all benchmarks. But by 1MB, 28.0% are multi-sharer entries, and at each benchmark's maximum PRD, 39.1% are multi-sharer entries.

Interestingly, this increase in sharing occurs non-uniformly. Figure 7 illustrates this by plotting the coverage for directory entries with 4 or more sharers (labeled "$\geq$ 4 sharers"). As Figure 7 shows, 2- and 3-sharer entries (*i.e.*, the gap between the $\geq$ 2 and $\geq$ 4 lines) 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,

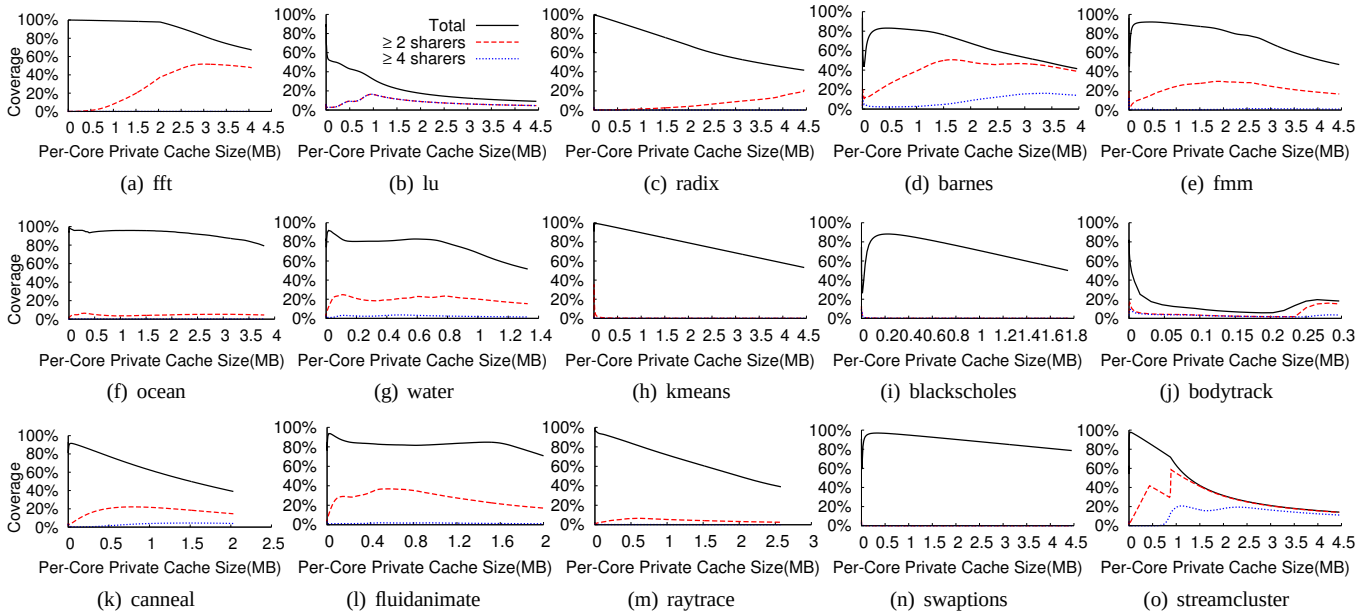


Fig. 7. Coverage *vs.* private data cache size for 64-core CPUs. Dashed lines break down single- *vs.* multi-sharer entries.

TABLE IV. COVERAGE FOR WIDELY SHARED ENTRIES, AND DROP IN COVERAGE DUE TO CORE COUNT SCALING.

Benchmark	≥32 Sharers	Coverage Drop 256KB	1MB
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%	11.8%	7.1%
ocean	0.003%	4.6%	4.1%
water	0.004%	14.1%	3.9%
kmeans	0.0002%	0.2%	-0.1%
blackscholes	0.02%	14.8%	1.8%
bodytrack	1.2%	8.9%	10.5%
canneal	0.02%	1.9%	0.4%
fluidanimate	0.0008%	2.3%	10.3%
raytrace	0.005%	1.8%	6.9%
swaptions	0.009%	6.2%	1.3%
streamcluster	0.04%	0.6%	-0.3%
Average	0.01%	2.6%	1.8%

Figure 8 plots additional lines for two of our benchmarks, showing entries with very wide sharing degree (note, the Y-axis is now on a log scale). As Figure 8 shows, directory entries with $\geq$ 32 sharers account for less than 0.01% coverage across most cache sizes, and do not increase with cache size scaling. We find this behavior is ubiquitous. In the second column of Table IV, we report the coverage for the same $\geq$ 32 entries for all 15 benchmarks at a private cache size of 1MB. Except for lu, Table IV shows $\geq$ 32 entries never account for more than 1.2% coverage. *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.*

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

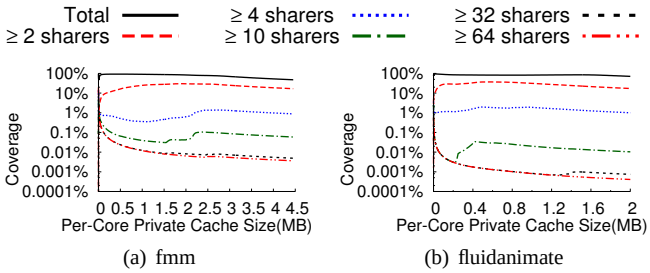


Fig. 8. Coverage breakdown for entries with wide sharing.

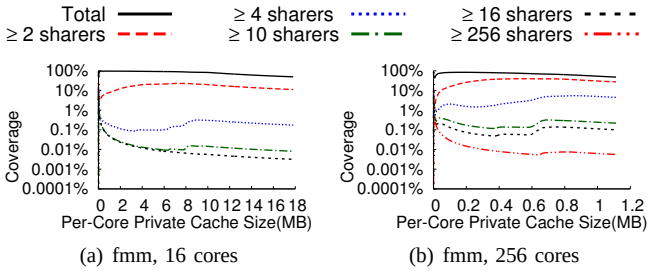


Fig. 9. Same as Figure 8(a) except for 16 and 256 cores.

needed for the directory. Somewhat surprisingly, though, the impact from core count scaling is much less. The last two columns of Table IV report the drop in coverage when scaling from 64 to 256 cores observed at private cache sizes of 256KB and 1MB, respectively. In most cases, coverage changes by only a few percent. For a few cases, the reduction can exceed 10%, but it does not approach the 2x shown in Figure 7.

Core count scaling's impact on coverage is limited because it tends to increase sharing for data that are widely shared. To illustrate, Figures 9(a)-(b) plot the same coverage breakdown from Figure 8(a), except the number of cores is 16 and 256. As Figure 9 shows, the coverage for widely shared entries increases with core count, but the entries with 2–3 sharers

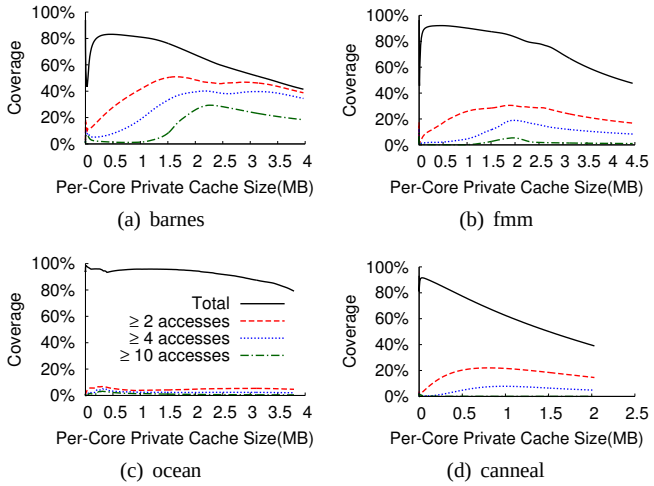


Fig. 10. Coverage breakdown by access count.

does not change much. Since the former is a small part of the directory cache, the overall impact on coverage is small.

C. Study 3: Access Distribution Results

Finally, we study the distribution of accesses over individual directory entries. We observe that the accesses an entry receives is related to its sharing degree. Entries with a few sharers tend to receive a few accesses (during their lifetimes) whereas entries with wide sharing tend to receive many accesses. Recall in Section V-B 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 few sharers. *So, like sharing, reuse of directory entries is also non-uniform, with a small part of the directory receiving a disproportionately large fraction of the directory accesses.*

Figure 10 helps illustrate 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 (as reported by the access counters in our profiler). In particular, we plot the coverage for entries with ≥ 1 (labeled "Total"), ≥ 2, ≥ 4, and ≥ 10 cache miss-induced accesses for a 64-core CPU. To save space, 4 representative benchmarks are shown.

Notice, the graphs in Figure 10 are similar to their counterparts in Figure 7. For instance, the breakdown for single-access entries (the gap between the "Total" and "≥ 2 accesses" lines in Figure 10) is essentially identical to the breakdown for single-sharer entries (the corresponding gap in Figure 7). Also, the breakdowns for the multi-access entries in Figure 10 are quite similar to those for the multi-sharer entries in Figure 7. Hence, many accesses are concentrated on a small number of directory entries.

Table V shows this leads to high reuse of the multi-access directory entries. In particular, columns 2–4 of Table V report the portion of accesses destined to entries with ≥ 3 accesses during their lifetimes. Results are shown for 256KB, 1MB, and ∞ private caches. We also report in columns 5–7 the portion of all directory entry lifetimes that the ≥ 3 lifetimes represent.

TABLE VI. DATA AND DIRECTORY CACHE SIZE SCALING PARAMETERS.

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

At 256KB, Table V shows the entries with ≥ 3 accesses account for only 5.4% of all directory entry lifetimes; yet, they receive 18.1% of all directory accesses (on average). At 1MB, they account for 23.0% of lifetimes, but receive 37.8% of the accesses. And at ∞, they account for 35.3% of lifetimes, but receive 54.9% of the accesses.

While these results include all accesses (T1 + T2), we find the reuse of multi-access entries is even greater for just the T2 transactions. In the last two columns of Table V, we report the portion of T2s destined to entries with ≥ 3 accesses for 256KB and 1MB private caches. These results show the great majority of T2s, 84.3% and 82.7%, are captured by a minority of directory entry lifetimes, 5.4% and 23.0%, respectively. As discussed in Section II, T2s are on-chip transactions whereas T1s can be off-chip transactions; hence, T2-induced directory accesses are more latency sensitive. Our results show *a small fraction of the directory cache can service the majority of latency-sensitive directory accesses.*

As discussed in Section I, recent directory designs have proposed asymmetric storage techniques [20], [28]. The above study suggests there exists asymmetry in temporal reuse over different directory entries that such techniques can exploit.

VI. VALIDATION AND DISCUSSION

This section conducts cache simulations to validate results from Section V. It also discusses the implications of our insights for existing directory techniques. After describing methodology, we present validations and discussion for the main insights in Section V (*i.e.*, from studies 1 and 2).

A. Simulation Methodology

We implemented a cache simulator that models the cache hierarchy in Figure 1 (without the shared cache). Our simulator uses the same PIN tool from Section IV except the LRU stacks are replaced by data cache models, and a directory cache model is added. In the data cache model, we employ three levels of private cache–an L1, L2, and L3 per core. The private caches are inclusive, and maintain coherence via a directory-based MESI protocol. All data caches use 64-byte blocks.

In the directory cache model, we implement Cuckoo [14]. Cuckoo directories use 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). They minimize the over-provisioning needed to mitigate conflicts at the expense of more costly insertions. In our Cuckoo directory, we assume full-map directory entries. This mirrors the precise tracking of sharers in our profiler. Although we

TABLE V. PERCENT ACCESSES DESTINED TO $\geq$ 3-ACCESS ENTRIES, PERCENT ENTRIES WITH $\geq$ 3 ACCESSES, AND PERCENT T2 ACCESSES DESTINED TO $\geq$ 3-ACCESS ENTRIES.

Benchmark	% Accesses to $\geq$3 Entries			% Entries with $\geq$3 Accesses			% T2 Accesses to $\geq$3 Entries	
	256KB	1MB	∞	256KB	1MB	∞	256KB	1M
fft	0.6%	4.4%	84.8%	0.0%	0.0%	71.1%	70.8%	10.3%
lu	54.1%	98.0%	98.4%	8.2%	51.0%	50.9%	100%	100%
radix	1.0%	2.2%	53.5%	0.1%	0.1%	25.9%	91.3%	55.0%
barnes	23.5%	54.3%	99.2%	12.3%	35.1%	92.5%	88.2%	86.6%
fmm	11.1%	31.7%	63.8%	5.2%	14.9%	27.0%	72.7%	76.4%
ocean	12.3%	25.4%	41.2%	5.2%	3.3%	4.9%	95.5%	97.5%
water	16.6%	49.9%	52.2%	6.3%	17.6%	17.6%	61.6%	83.8%
kmeans	0.1%	0.1%	0.1%	0.5%	0.1%	0.1%	100%	100%
blackscholes	0.8%	0.8%	0.8%	0.1%	0.0%	0.0%	100%	100%
bodytrack	68.5%	98.7%	98.7%	11.4%	85.2%	85.2%	99.7%	100%
canneal	19.5%	46.7%	54.6%	7.5%	27.1%	30.1%	65.5%	88.8%
fluidanimate	42.3%	64.6%	70.1%	23.4%	35.5%	23.3%	89.1%	98.6%
raytrace	4.1%	7.1%	7.1%	0.8%	0.6%	0.5%	59.7%	43.4%
swaptions	0.1%	0.1%	0.1%	0.0%	0.0%	0.0%	100.%	100%
streamcluster	16.3%	82.4%	98.8%	0.2%	73.9%	98.4%	70.0%	99.9%
Average	18.1%	37.8%	54.9%	5.4%	23.0%	35.3%	84.3%	82.7%

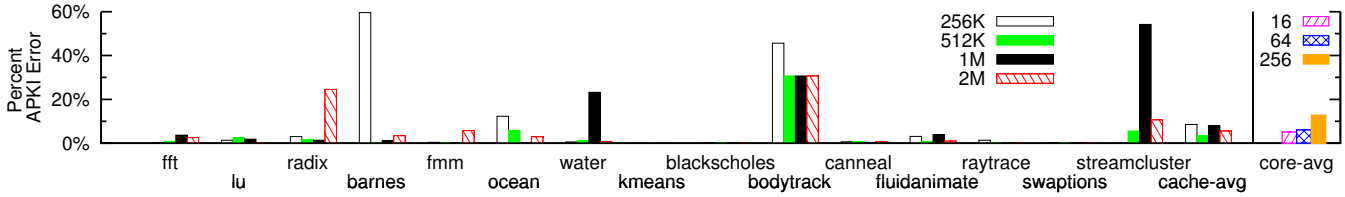


Fig. 11. Percent APKI error for cache miss-induced (T1+T2) directory accesses.

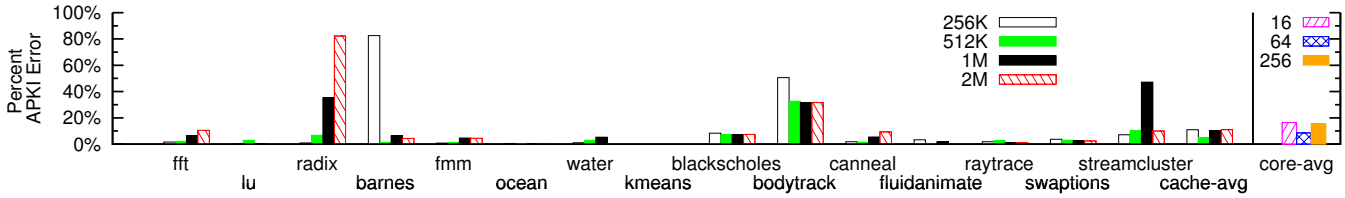


Fig. 12. Percent APKI error for T2 directory accesses.

validate against full-map, our results have implications for other implementations as well (see Section VI-C).

We perform simulations that scale the data cache hierarchy across both capacity and core count. Capacity scaling is performed on the L3 cache across four different sizes at 64 cores. The top portion of Table VI specifies the cache parameters used. Core count scaling is performed across three different core counts–16, 64, and 256–at a total L3 capacity of 64MB (using 4MB, 1MB, and 256KB per-core L3s). We also perform simulations that scale the directory size. The bottom portion of Table VI specifies the sizes in terms of coverage. (Different numbers of memory arrays, *i.e.*, "ways" in Table VI, are used to maintain a power-of-2 number of sets as coverage varies).

B. Study 1

Validation. Our framework analyzes capacity misses and sharing, but not other cache effects, like conflicts. Figures 11 and 12 quantify the error this introduces into our directory access results from Section V-A. In the figures, we plot the error between the simulated and profiled directory APKI. The per-benchmark bars report this error for cache size scaling at 64 cores (*i.e.*, for the L3 cache sizes in Table VI). Figure 11 shows the error for cache miss-induced directory accesses while Figure 12 shows the error for T2 accesses (the "Total Misses" and "T2" lines, respectively, from Figure 5).

As Figures 11 and 12 show, the simulator and profiler are very close in most cases. For 83% (75%) of the data points in Figure 11 (Figure 12), the profiler is within 7% of simulation. Averaged across all 64-core validations, the error in total cache miss-induced (T2) APKI is 6.3% (9.3%), as shown by the "cache-avg" bars. (We also find similar accuracy results when including notifications, but omit them to save space). The main reason for these errors is conflict misses in the cache simulations which our profiler does not model since it assumes full associativity.

Several validation points in Figures 11 and 12 have high error, but in fact, most of these are benign. As discussed in Section V-A, cache size scaling reduces directory access frequency, making APKI very small for some benchmarks and cache sizes. In these cases, tiny absolute errors can result in large percent error. This happens in bodytrack and radix's T2 accesses. Also, access frequency can drop suddenly (*e.g.*, Fig-

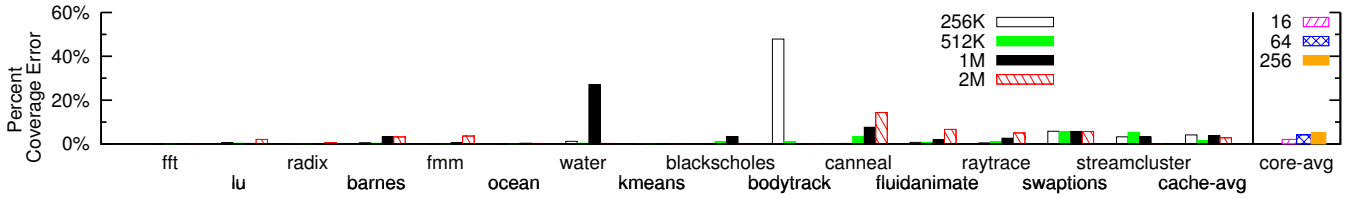


Fig. 13. Percent coverage error.

ure 5(d)). If the profiler slightly mis-judges the capacity where such drops occur, large errors can accrue, but only locally around the drop. This happens in barnes and streamcluster.

We also validate our core count scaling analyses. In Figures 11 and 12, the last set of bars labeled "core-avg" report the error for core count scaling at 64MB of total L3 cache, *i.e.*, for 16, 64, and 256 cores, averaged across all benchmarks (we omit per-benchmark results). The error in total cache miss-induced (T2) APKI is 8% (13%). Like cache size scaling, a few benign cases also account for most of the error in core count scaling. Overall, Figures 11 and 12 show our framework predicts directory cache accesses quite accurately, especially when considering many "bad" cases are benign.

Discussion. The down-stream traffic from a cache reduces as its size goes up. This is well understood for data caches, especially in uniprocessors. Our analyses in Sections III-B and V-A show how to 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 total accesses drop with CPU scaling, especially cache size scaling–implies directory cache accesses will make up a smaller fraction of overall execution time as CPUs scale. Among the directory designs discussed in Section I, many propose techniques for reducing directory size that also increase the cost of directory lookups–*e.g.*, due to more complex hash functions (tagless directories [35]) or multiple accesses (hierarchical directories [15]). 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 V-A 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 re-insertion techniques–for example, Cuckoo [14] and SCD [25]. 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 significant 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.

C. Study 2

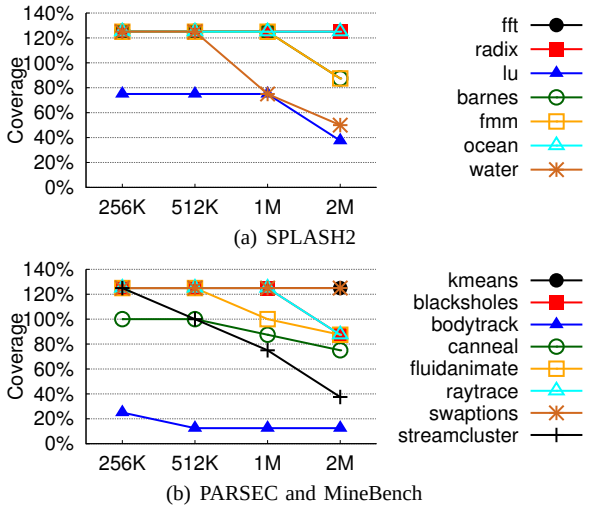
Validation. In addition to directory accesses, there is also error in our directory contents analyses from Section V-B. In particular, Figure 13 quantifies the error in our coverage results from Figure 7. We ran our data and directory cache simulator using the largest Cuckoo directory in Table VI–200% coverage–and measured the average number of live directory entries in the simulated directory cache. (In Cuckoo, 200% over-provisioning ensures virtually no entries are evicted due to conflicts). The per-benchmark bars in Figure 13 plot the error between the simulated and profiled coverage for cache size scaling at 64 cores (*i.e.*, for the L3 cache sizes in Table VI).

Figure 13 shows our framework accurately predicts coverage. Averaged across all 64-core validations, the coverage error is only 3.0%, as shown by the "cache-avg" bars. Error is high in one datapoint for bodytrack, but as in Section VI-B, this is due to a tiny absolute error being compared to a very small simulated coverage. The remaining cases reflect the error due to our framework's inability to account for conflicts in the private data caches, as was discussed in Section VI-B.

The bars labeled "core-avg" in Figure 13 report the coverage error for core count scaling at 64MB of total L3 cache averaged across all 15 benchmarks (we again omit the per-benchmark results). Figure 13 shows the error in coverage across core count scaling is similar to cache scaling, 3.7% on average for all the datapoints.

Discussion. A very basic design question is how large should the directory cache be? Our analyses in Sections III-C and V-B can help architects answer this question. In Figure 7, we show coverage reduces with CPU scaling, especially data cache size scaling. This implies that the fraction of on-chip memory devoted to the directory cache can be reduced as CPUs scale without impacting performance. We ran cache simulations to test this result. For a 64-core CPU and for each of our L3 data cache sizes, we simulated all of the Cuckoo sizes in Table VI and identified the minimum that effectively caches all directory entries. (We require the fraction of directory entry insertions that evict a live entry to be less than 1%). In Figure 14, we plot these minimum Cuckoo sizes (in terms of coverage) across private data cache size.

Figure 14 confirms the Cuckoo directory's coverage drops with data cache size scaling. Most benchmarks require 125% coverage at 256KB private caches. But as private caches scale to 2MB, only 5 benchmarks remain at 125%. Six benchmarks drop to 75–87.5% coverage, while 4 benchmarks drop to 50% or less. Moreover, comparing Figures 14 and 7, we see the minimum Cuckoo sizes are correlated to the profiled coverage. In most cases, the Cuckoo coverage is between 30–50% higher than the profiled coverage, which quantifies the over-provisioning needed in the directory cache to mitigate conflicts. These results show directory coverage indeed varies significantly with architectural scaling, and that our framework can help identify the minimum directory size for each bench-



(a) SPLASH2

(b) PARSEC and MineBench

Fig. 14. Minimum Cuckoo coverage for 1% eviction rate.

mark.

In Figure 14, we assume full-map entries which matches our profiler's precise sharer tracking. But many sharer lists are possible. For instance, hierarchical lists [15], [25], [29] avoid allocating many sharer bits. Our framework can handle hierarchical lists, but a modification to the profiler is needed to account for root and leaf entries. Many techniques use imprecise sharer lists that incur extra coherence messages–*e.g.*, limited pointers with broadcast [2] and coarse vectors [16]. They do not affect the tags stored in the directory, so our existing profiler can already handle them. Unfortunately, imprecise sharer lists that incur directory-induced invalidations–*e.g.*, limited pointers with invalidation [2]–cannot be analyzed in an architecture independent fashion. So, our current framework cannot track their directory's contents.

Finally, another observation from Section V-B affecting directory size is that the mix of private and shared directory entries changes with CPU scaling, especially data cache size scaling. A popular approach used recently, as discussed in Section I, is to compact the storage for private entries [3], [28], [34]. Our results show these techniques can be very effective; however, their actual impact is highly sensitive to scaling. In particular, Figure 7 corroborates previous claims that the majority of directory entries encode a single sharer [11]. For example, on average, 85.5% of entries are private for 256KB data caches. But the private entries reduce to 72.0% at 1MB, and at each benchmark's maximum PRD, the private entries reduce to 60.9%. So, the opportunity for these techniques varies significantly. Interestingly, the curves labeled "$\geq$ 2 sharers" in Figure 7 quantify the best these techniques can ever do (*i.e.*, if storage for private entries goes to zero). In the limit, removing private entries can lower coverage a lot (compared to the "Total" curves). But the gain is not as large when CPUs scale since the $\geq$ 2 sharer curves increase due to greater sharing.

VII. Related Work

The difficulty of simulating large CPUs has motivated researchers to develop alternative evaluation methodologies. In particular, significant research has explored RD analysis techniques. Our work exploits PRD profiling [27], [26], [33], [32] which evaluates private caches. There has also been work on evaluating shared caches [12], [19], [31], [33], [32]. But all of these efforts have focused on data caches. This paper is the first to apply reuse distance for reasoning about directory caches, and for studying their behavior under CPU scaling.

Besides RD analysis, researchers have also developed analytical models. For example, the Cuckoo work [14] used a model to estimate the directory's size and energy as core count and data cache size are simultaneously scaled. In addition, SCD [25] presented a model for estimating over-provisioning in the directory cache. Compared to RD analysis, analytical models are much faster. However, analytical models for directory caches require making assumptions about workload behavior and its interactions with the data caches, so they are not as accurate as RD-based techniques.

Finally, there is a large body of research on directory caches (see Section I), and some have studied the effects of multicore scaling on their techniques using simulation. Most only simulate a single core count and cache size [11], [10], [15], [16], [25], [28], [35], [34]. Cuckoo and SCT [3] simulated 2 different private cache sizes at a single core count. WayPoint [20] 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.

VIII. Conclusion

In the past, reuse distance has been employed to analyze I/O sub-systems, virtual memory, as well as processor data caches (both sequential and parallel). This paper takes the next step for RD analysis: applying it to analyze multicore directory caches and extracting 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. Although we examine the hardware implications of our insights, there may be other benefactors. We believe compilers can use our analyses for optimization–*e.g.*, tuning sharing to change a program's footprint in the directory cache. Also, performance programmers can potentially use our techniques to identify scaling bottlenecks. We hope future advances can come from our work.

Acknowledgments

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

References

[1] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff, "Tile Processor: Embedded Multicore for Networking and Multimedia," in *Proceedings of the 19th Symposium on High Performance Chips*, Starford, CA, USA, 2007.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," in *Proceedings of the 15th International Symposium on Computer Architecture*, Los Alamitos, CA, USA, 1988.

[3] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *Proceedings of the 45th Annual International Symposium on Microarchitecture*, Washington, DC, USA, 2012.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, 2000.

[5] E. Berg, H. Zeffer, and E. Hagersten, "A statistical multiprocessor cache model," in *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, TX, USA, 2006.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, 2008.

[7] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 1991.

[8] G. Chen, "SLiD–A Cost-Effective and Scalable Limited-Directory Scheme for Cache Coherence," in *Proceedings of the '93 Parallel Architectures and Languages Europe*, Heidelberg, Germany, 1993.

[9] J. H. Choi and K. H. Park, "Segment Directory Enhancing the Limited Directory Cache Coherence Schemes," in *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 1999.

[10] B. Cuesta, A. Ros, M. E. Gomez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks," *IEEE Transactions on Computers*, vol. 62, no. 3, 2013.

[11] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proceedings of the 38th annual international symposium on Computer architecture*, New York, NY, USA, 2011.

[12] C. Ding and T. Chilimbi, "A Composable Model for Analyzing Locality of Multi-threaded Programs," Microsoft Research, Technical Report MSR-TR-2009-107, 2009.

[13] D. Eklov, D. Black-Schaffer, and E. Hagersten, "Fast modeling of shared caches in multicore systems," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, New York, NY, USA, 2011.

[14] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, San Antonio, TX, USA, 2011.

[15] S.-L. Guo, H.-X. Wang, Y.-B. Xue, C.-M. Li, and D.-S. Wang, "Hierarchical Cache Directory for CMP," *Journal of Computer Science and Technology*, vol. 25, no. 2, 2010.

[16] A. Gupta, W. dietrich Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Proceedings of the 1990 International Conference on Parallel Processing*, Urbana-Champaign, IL, USA, 1990.

[17] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, "Exploring the Cache Design Space for Large Scale CMPs," *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.

[18] Intel, "Intel Xeon Phi Product Family," 2014.

[19] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?" in *Proceeding of the 2010 Compiler Construction*, Paphos, Cyprus, 2010.

[20] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel, "WayPoint: Scaling Coherence to 1000-core Architectures," in *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*, New York, NY, USA, 2010.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005.

[22] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, 1970.

[23] C. McCurdy and C. Fischer, "Using pin as a memory reference generator for multiprocessor simulation," *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.

[24] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads," in *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*, San Jose, CA, USA, 2006.

[25] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, New Orleans, LA, USA, 2012.

[26] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria, 2010.

[27] D. L. Schuff, B. S. Parsons, and J. S. Pai, "Multicore-Aware Reuse Distance Analysis," Purdue University, Technical Report TR-ECE-09-08, 2009.

[28] J. J. Valls, A. Ros, J. Sahuquillo, M. E. Gómez, and J. Duato, "Ps-dir: a scalable two-level directory cache," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, New York, NY, USA, 2012.

[29] D. A. Wallach, "PHD: A Hierarchical Cache Coherent Protocol (Master's Thesis)," 1993.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995.

[31] M.-J. Wu and D. Yeung, "Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs," in *Proceeding of the 20th International Conference on Parallel Architectures and Compilation Techniques*, Galveston Island, TX, USA, October 2011.

[32] M.-J. Wu and D. Yeung, "Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs," *ACM Transactions on Computer Systems*, vol. 31, no. 1, 2013.

[33] M.-J. Wu, M. Zhao, and D. Yeung, "Studying Multicore Processor Scaling via Reuse Distance Analysis," in *Proceeding of the 40th International Symposium on Computer Architecture*, Tel-Aviv, Israel, June 2013.

[34] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-Grain Coherence Directories," in *Proceedings of the 46th Annual International Symposium on Microarchitecture*, Davis, CA, USA, 2013.

[35] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A Tagless Coherence Directory," in *Proceedings of the 42nd International Symposium on Microarchitecture*, New York, NY, USA, 2009.

[36] L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell, "Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design," in *Proceedings of the 1st Workshop on Chip Multiprocessor Memory Systems and Interconnect*, Phoenix, AZ, USA, 2007.