Studying Directory Access Patterns via Reuse Distance Analysis and Evaluating Their Impact on Multi-Level Directory Caches

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

Abstract

The trend for multicore CPUs is towards increasing core count. One of the key limiters to scaling will be the on-chip directory cache. Our work investigates moving portions of the directory away from the cores, perhaps to off-chip DRAM, where ample capacity exists. While such multi-level directory caches *exhibit increased latency, several aspects of directory accesses will shield CPU performance from the slower directory, including low access frequency and latency hiding underneath data accesses to main memory.*

While multi-level directory caches have been studied previously, no work has of yet comprehensively quantified the directory access patterns themselves, making it difficult to understand multi-level behavior in depth. This paper presents a framework based on multicore reuse distance for studying directory cache access patterns. Using our analysis framework, we show between 69–93% of directory entries are looked up only once or twice during their liftimes in the directory cache, and between 51–71% of dynamic directory accesses are latency tolerant. Using cache simulations, we show a very small L1 directory cache can service 80% of latency critical directory lookups. Although a significant number of directory lookups and eviction notifications must access the slower L2 directory cache, virtually all of these are latency tolerant.

1. Introduction

The trend for high-performance CPUs is towards integrating a larger number of cores on-chip. As core count increases, scalability will become a major issue. Many factors can limit multicore scalability, but one of the key culprits is the cache hierarchy and its associated coherence hardware. Maintaining cache coherence across a large number of cores requires directory-based coherence protocols that employ directory caches [14]. Unfortunately, these structures can become very large, consuming significant area and power.

The problem is directories tend to increase as a function of both data cache size and core count, resulting in superlinear growth as multicores scale. Numerous techniques have tried to address this problem, including minimizing sharing vectors [1, 5, 7, 8, 9, 14, 22, 25] as well as reducing the number of directory entries by mitigating over-provisioning [13] or omitting entries for private data [11, 10]. Most techniques, however, do not eliminate the superlinear growth problem, and those that can achieve linear scaling do so at the expense of being able to efficiently track data blocks for certain sharing patterns, degrading performance.

Rather than compact the directory, this paper investigates varying its proximity to the cores. Most existing techniques implement directory caches monolithically in SRAM on the same die as the cores and data caches. This enables high-speed access to both data and directories. While efficient access to on-chip data is essential for performance, it is less clear whether the same is true for directories. If parts of the directory are insensitive to slower access speeds, they could be stored farther away from the cores, perhaps even in DRAM. Such *multi-level directory caches* would provide more flexibility to increase directory size as CPUs scale.

In fact, researchers have recently investigated this approach for improving directory scaling. In particular, PS-DIR [25] proposes using embedded DRAM (eDRAM) to provide higher density directory storage on the CPU die itself. Alternatively, WayPoint [17] uses system DRAM to implement an off-chip extension of the on-chip directory cache.

For such multi-level designs to succeed, it should be the case that CPU performance is less sensitive to the latency of directory cache access compared to data cache access. There are several reasons to believe this is true. First, directory accesses are typically *infrequent* since they occur along the miss path of the private data caches. Because hits in cores' private caches usually dominate the misses, a large portion of the CPU's memory references are "filtered" from the directory cache, reducing its overall access rate.

But also, a significant number of directory entries may be accessed very rarely. This is due to *private data*. A data block that is only referenced by a single core does not generate any directory accesses after it is filled into the cache. Upon eviction, it may generate a single notification to the directory. Hence, the associated directory entry could receive one or two accesses during its entire lifetime in the directory cache. Researchers have observed private data dominate in many parallel programs [11, 10, 25], so the majority of directory entries may exhibit such low temporal reuse.

Not only are directories accessed infrequently, but many accesses are *latency tolerant*. In particular, the directory accesses associated with data cache fills may be performed in parallel with data accesses to main memory. In these cases, the directory access latency may be hidden underneath the long latency data access. Moreover, data cache evictions that notify the directory can be performed after their data cache fills complete. In these cases, the directory access latency is completely off the CPU's critical path.

These observations suggest major portions of the directory cache can be moved to slower memory for scalability, and indeed, the recent research has shown promising results [17, 25]. However, no previous work has comprehensively studied the directory access patterns themselves, which can be quite complex. In particular, the cache misses that determine directory accesses depend on complex interactions between applications' memory accesses and the data cache hierarchy. Worse yet, these directory accesses vary with data cache size. Not only does the number of misses change, but the amount of sharing captured between data caches also changes, affecting the private cases mentioned above. Without detailed studies that shed light on such complex access patterns, it is difficult to understand multi-level behavior.

Traditionally, computer architects have used architectural simulation alone to study cache effects. Simulators can model memory behavior accurately, but they provide very limited insights. Deep insights usually require exploring numerous cache configurations and observing how architecture-application interactions change. Due to their slow speed, simulators can only consider a moderate number of configurations. But as multicores scale, their cache design spaces can easily exhibit 1000s to millions of different configurations.

Recently, there has been significant interest in evaluating multicore cache hierarchies via locality analysis [12, 16, 24, 23, 28, 29]. These techniques acquire *reuse distance (RD) profiles* using LRU stacks to characterize program-level locality. The key is profiles are architecture independent, so a few profiles can reveal caching behavior across entire design spaces. Recent advances have extended uniprocessor profiling to handle multicore CPUs by modeling *inter-thread interactions*. For example, private-stack reuse distance (PRD) profiling [24, 23, 28] uses per-thread coherent LRU stacks to model the interactions that occur in private data caches.[1] PRD (and other techniques) have already yielded significant insights that have eluded simulation studies.

In this paper, we study multi-level directory caches using two evaluation methodologies. First, we employ multicore RD analysis to study directory cache access patterns. To enable this study, we propose a framework based on PRD stacks that can extract the directory access stream associated with directory lookups, and implement it in a profiler. A key feature of our framework is its ability to analyze *relative reuse distance between sharers*, thus assessing sharing in a capacity-sensitive fashion. Our framework can quantify the frequency and distribution of lookups across the directory state. It can also break down the number of latency tolerant accesses. Last but not least, our framework can perform all of these analyses at every possible private data cache size, revealing how directory access patterns will evolve with data cache scaling.

[1] PRD is sensitive to memory reference interleaving, so strictly speaking, it is architecture dependent. But studies have shown interleaving perturbations are benign for *symmetric threads* in programs exploiting loop-level parallelism [16, 28]. So, PRD profiles are accurate for this class of programs.

Second, we use cache simulation to quantify the caching performance of directories with two levels of hierarchy. Our simulator evaluates how effectively the L1 directory cache captures temporal reuse, especially for directory accesses that are latency sensitive. It also measures the accesses to the L2 directory cache, including writeback notifications that our analysis framework does not model. Most importantly, we cross-reference our analyses and simulations to check for agreement between the two methodologies.

Our profiler shows the majority of directory entries are accessed infrequently. Across a suite of 15 parallel benchmarks running on 64 cores, we find between 69–93% of directory entries (depending on private cache size) receive only 1 or 2 lookups during their lifetimes in the directory cache. And most, between 62–86%, receive only a single lookup. Relatively few directory entry lifetimes (7–31%) witness more than two lookups. In addition, our analysis framework also shows the majority of directory cache accesses, between 51–71%, are latency tolerant. Comparatively fewer lookups, between 29–49%, are latency critical.

Our simulator shows multi-level directory caches can effectively exploit the temporal reuse on directory entries shown by our analyses, especially for latency critical lookups. We find an L1 directory cache with only 4.5% of the private data caches' SRAM (*i.e.* providing 0.18X coverage of the data cache blocks) can achieve a 75–80% hit rate for latency critical directory lookups. A significant fraction of all directory lookups, between 58–74%, still miss in the L1 directory cache, resulting in L2 directory cache traffic of between 2.3–4.2 accesses per 1000 instructions (APKI). However, the vast majority of these are latency tolerant. Finally, writeback traffic to the L2 directory cache is significant too, between 2.6–6.5 APKI. But all of these are latency tolerant as well.

The rest of this paper is organized as follows. Section 2 discusses directory cache accesses, and their implications for multi-level implementation. Then, Section 3 presents our framework for analyzing directory cache access patterns. Next, Sections 4 and 5 report results from our analysis framework and cache simulations, respectively. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2. Directory Cache Accesses

Figure 1 illustrates a multicore cache hierarchy. 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). Below the private caches is the CPU's *sharing point* where the directory cache sits. Optionally, there may also be a shared data cache at the sharing point. Finally, main memory appears below the directory and shared data caches.

The directory cache is accessed on data cache transactions that perform directory lookups. It is also accessed when data cache evictions notify the directory. Our analysis framework focuses on the lookup-inducing cache transactions (the notifications are addressed by our simulations). To illustrate, Fig-

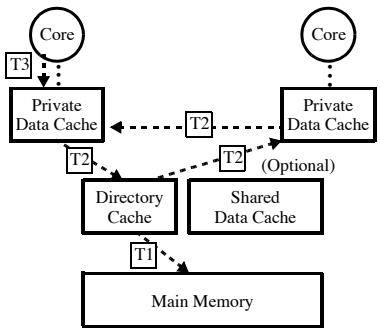

Figure 1: Different directory cache accesses.

ure 1 groups cache transactions into 3 categories, labeled T1–T3, based on a directory entry's life cycle. First, a memory request may miss all the way to main memory (T1), causing a new data block to be brought on-chip and a directory entry fill into the directory cache. These transactions initiate *new directory entry lifetimes* within the directory cache. Second, a memory request may miss to the sharing point, but find its data on-chip in a remote private cache (T2). These "sharing-based" transactions require directory lookups to determine the kind of remote actions needed as well as the sharers involved. Their accesses *reuse directory entries previously filled by T1 transactions*. Third, a memory request may hit in a core's private data cache (T3). These transactions are satisfied completely within the core's local private cache hierarchy, and do not lookup the directory. Finally, the life cycle ends when all copies of a data block and the directory entry have been evicted from the data and directory caches.

Two characteristics of the cache transactions in Figure 1 determine a directory cache's impact on CPU performance. The first is access frequency. In particular, total access frequency–*i.e.* the number of T1 + T2 *vs.* T3 transactions–reflects the importance of directory cache accesses compared to data cache accesses. But in addition, per-entry access frequency–*i.e.* the distribution of T1 and T2 transactions across different directory entries–reflects the criticality of individual entries.

The second characteristic is latency tolerance. Because T2 transactions incur on-chip latencies and are serialized with their directory lookups (they need the sharing set to proceed), they are sensitive to directory access latency. In contrast, T1 transactions incur much higher off-chip latencies and only require the memory address to proceed. Although a directory lookup is still needed to determine the data is not on-chip, this can be speculated (see Section 2.1), allowing the lookup and data access to occur in parallel. So, T1 transactions are tolerant of slower directory access. The number of T2 *vs.* T1 transactions reflects the latency sensitivity of the directory cache.

Notice, the cache transactions in Figure 1 are determined by the private data caches. This means a directory cache's access frequency and latency tolerance characteristics *vary with the private data caches*. In particular, as private cache size scales, the number of T1 + T2 transactions will change as well as their distribution over different directory entries. Furthermore, private cache size scaling will also change the

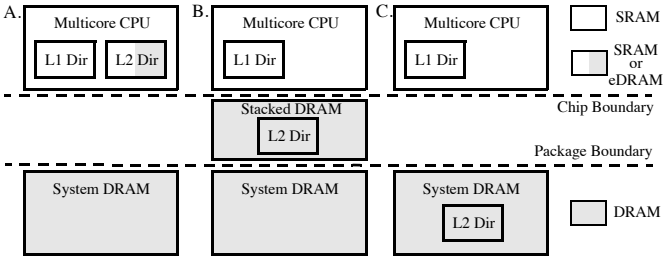

Figure 2: Multi-level directory cache implementations.

amount of sharing that is captured on-chip. This can turn shared accesses into private accesses, affecting the T2/T3 balance. But it can also change the number of misses that find their data on- *vs.* off-chip, affecting the T1/T2 balance and the directory cache's latency sensitivity.

Lastly, while the impact of directory caches on CPU performance is mainly determined by private caches, shared data caches also play an important role. For example, shared cache hits for data that would otherwise not be on-chip affect the latency sensitivity of some T1 transactions. They potentially make the associated directory lookups latency critical even though their data is not resident in the private caches. Section 5.3 will study shared caches, and evaluate techniques to address such latency sensitivity issues.

2.1. Implications for Directory Implementation

The different cache transactions in Figure 1 give rise to asymmetric directory access latency requirements. Conventional directory caches do not exploit such asymmetry. They treat all directory entries equally, storing them in SRAM on the CPU die for high performance. But this is overkill for many accesses. Our work studies directory caches with two levels–a fast *L1 directory cache* backed by a slower *L2 directory cache*–that exploit access asymmetry to improve scalability.

In particular, directory entries that are frequently involved in T2 transactions should be kept in the L1 directory cache. This will provide low latency to the most important entries with lookups on the CPU's critical path. In contrast, directory entries solely involved in T1 transactions can reside in the L2 directory cache. Due to T1's latency tolerance, keeping their entries in the slower L2 will have little impact on CPU performance, but will take capacity pressure off of the L1. It may also be possible to place directory entries with infrequent T2 transactions in the L2. This will further reduce L1 pressure while slowing down only a few latency-critical transactions.

Notice, if strong differentiation is achieved between the two directory caches, then T1 speculation can be highly accurate. As mentioned earlier, T1 transactions are (strictly speaking) serialized with their directory accesses. But if very few T2 transactions access the L2 directory cache, then an L1 miss would strongly suggest that the transaction is of type T1.

For high performance, the L1 should be implemented in SRAM on-chip, just like a conventional directory cache. But there are many possibilities for the L2, as shown in Figure 2. First, the L2 can also be implemented in on-chip SRAM. While this does not reduce the total on-chip directory size,

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

Figure 3: Two interleaved memory reference streams.

it permits a smaller L1, allowing L1 hits to incur lower latency and power. It also allows energy reduction techniques that increase access latency to be aggressively applied to the L2 only. Another on-chip solution is to implement the L2 in eDRAM. This is the approach taken by PS-Dir [25]. It can improve directory scaling since eDRAM enables a larger directory than SRAM given the same area.

Alternatively, the L2 directory cache can be implemented in off-chip DRAM, either as a stacked die on top of the CPU die or in main memory, as is done in WayPoint [17]. These two approaches essentially provide unlimited capacity to implement the L2, and hence, offer the greatest potential for directory scaling. However, they also increase the L2's access latency and energy. Fortunately, our results will show most L2 lookups are T1 transactions. Because T1s occur in parallel with data accesses to system DRAM, we expect the increased latency to have no performance impact for stacked DRAM and only minimal impact for main memory. But increased energy, especially for the main memory option, can be significant. Of the options in Figure 2, stacked DRAM offers the best scalability with minimal performance/power impact, **but all options are viable**.

3. Analysis Framework

This section presents our analysis framework. Section 3.1 reviews multicore RD techniques. Then, Section 3.2 develops new analyses to identify different cache transactions and their associated directory lookups, as discussed in Section 2.

3.1. Multicore RD Analysis

Reuse distance has been used to analyze uniprocessor locality. A reuse distance (RD) profile is a histogram of RD values for all memory references in a sequential program, where each RD value is the number of unique data blocks referenced since the last reference to the same data block. Because a cache of size *CS* can satisfy references with RD $< CS$ (assuming LRU), its cache misses can be predicted as the sum of all references in an RD profile above the RD value for capacity *CS*. Also, RD profiles 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 [24, 23, 28, 29] 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 hits and misses occurring within private data caches.

To illustrate, Figure 3 shows the memory references from two cores performed on data blocks *A*–*J*, and Figure 4 shows the corresponding LRU stacks at different times. In particular, Figure 4(a) shows C_1's re-reference of *A* at $t = 10$, assuming

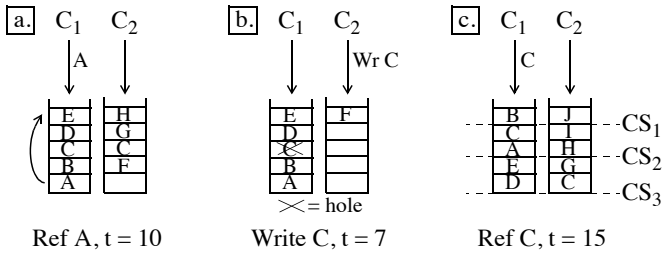

Figure 4: LRU stacks illustrating (a) intra-thread reuse and replication, (b) invalidation, and (c) PRD_{remote}.

all references in Figure 3 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. Similar to 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 3, both C_1 and C_2 access data block *C*. Assuming these are both reads, Figure 4(a) shows the *C* block is replicated in the cores' stacks. Such shared replicas increase capacity pressure in the affected stacks, thus modeling the additional cache misses that would occur.

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 Figure 4(b). To prevent promotion of blocks further down the LRU stack, invalidated blocks leave behind *holes* [24]. 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 4(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 = ∞.

3.2. Directory Access Analysis

Because PRD profiling can predict private data cache misses, it has the basic capability to identify directory cache lookups. In particular, given a memory reference's PRD and access mode (read or write), we can predict whether a directory access will occur at some cache size *CS*, and if so, its type.

Consider the examples from Section 3.1. In Figure 4(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 lookup occurs. Otherwise (PRD $\geq CS$), the reference misses and generates a directory lookup. Given there are no other copies of *A* on-chip, this is a T1 transaction that initiates a new directory entry lifetime. In Figure 4(b), if C_2's reference to block *C* is a write rather than a read, then the references at $t = 7$ and 12 would both

miss and generate directory lookups. (Like Figure 4(a), these also depend on the cache size CS, which we address below). Since these are due to inter-thread communication, they are T2 transactions that reuse the directory entry filled 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 a program's threads share data, whether or not that sharing manifests itself on-chip depends on the size of the CPU's caches. So, the frequency of sharing-based T2 transactions is also tied to temporal locality–in particular, to the *relative reuse distance between sharers*.

Figure 4(c) illustrates this by showing C_1's reuse of block C at $t = 15$. Both cores have brought the block into their LRU stacks, but C_1 has referenced the block more recently than C_2. So, the block appears 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 4(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 lookup. If the cache size is CS_2, then only C_1's copy is on-chip. We say block C is "temporally private"–*i.e.* it is private within the limited time window captured by CS_2. In this case, C_1's reference is a hit regardless of access mode (a T3 transaction) with no directory lookup. 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 sharing-based T2 directory lookup.

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 (*i.e.* all stacks except for the core's local stack). 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, however, 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, $\text{PRD}_{remote} \geq CS$ would mean the memory reference is temporally private.

Table 1 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 1 reports all of them in terms of the T1–T3 categories.

3.2.1. Access Mode, PRD, PRD_{remote} Characterization. The first eight transactions in Table 1 form the T1 category. All of these do not find the requested block in the local private cache, and there is no sharing captured on-chip (PRD and $\text{PRD}_{remote} \geq CS$). Transactions 1 and 2 correspond to Figure 4(a) assuming $CS < 5$; transactions 3 and 4 correspond to Figure 4(c) assuming $CS = CS_1$; transactions 5 and 6 represent cold misses; and transactions 7 and 8 are indistinguishable from cold misses, though a remote core did previously access the block. Because the requested data is not on-chip, these transactions perform off-chip accesses.

	Mode	PRD	PRD_{remote}	Comment
T1 Transactions: New Lifetimes				
1	R	$\geq CS$	∞	Truly Private
2	W	$\geq CS$	∞	Truly Private
3	R	$\geq CS$	$\geq CS$	Temporally Private
4	W	$\geq CS$	$\geq CS$	Temporally Private
5	R	∞	∞	Cold Miss
6	W	∞	∞	Cold Miss
7	R	∞	$\geq CS$	Temporal Cold Miss
8	W	∞	$\geq CS$	Temporal Cold Miss
T2 Transactions: Directory Reuse				
9	R	∞	$< CS$	Coherence
10	W	∞	$< CS$	Coherence
11	R	$\geq CS$	$< CS$	Forwarding
12	W	$\geq CS$	$< CS$	Coherence
13	W	$< CS$	$< CS$	Coherence
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

Table 1: Access mode, PRD, and PRD_{remote} characterization of data cache transactions and T1–T3 categorization.

The next five transactions in Table 1 form the T2 category. All of these exhibit sharing that is captured on-chip ($\text{PRD}_{remote} < CS$) and some remote action is required–either invalidation or forwarding of the requested block. Transaction 10 corresponds to Figure 4(b) assuming $PRD_{remote} < CS$, and transaction 9 corresponds to the same example, but later at $t = 12$. Transactions 12 and 13 are similar to 10 and transaction 11 is similar to 9 except the accessed block was referenced by the local core previously, but we still need remote actions. Because remote actions are needed, these transactions' directory lookups are on the CPU's critical path.

The last five transactions form the T3 category. All of these exhibit sufficient temporal reuse to be captured in the local private cache (PRD $< CS$) and do not require remote actions. Transactions 14 and 15 correspond to Figure 4(a) assuming $CS \geq 5$; transactions 16 and 17 correspond to Figure 4(c) assuming $CS = CS_2$; and transaction 18 corresponds to Figure 4(c) assuming $CS = CS_3$. Because these transactions are satisfied locally, they do not incur directory lookups.

4. Access Pattern Experiments

This section applies our analysis framework to gain insights into directory cache access patterns. First, we describe our profiler that implements the analysis framework. Then, we present the profiling results.

4.1. PIN Profiler

We implemented directory access profiling within the Intel PIN tool [18]. We modified PIN to maintain coherent private LRU stacks and perform PRD profiling, as discussed in Section 3.1. (We assume 64-byte blocks in all LRU stacks). For every memory reference, our profiler consults the LRU stacks

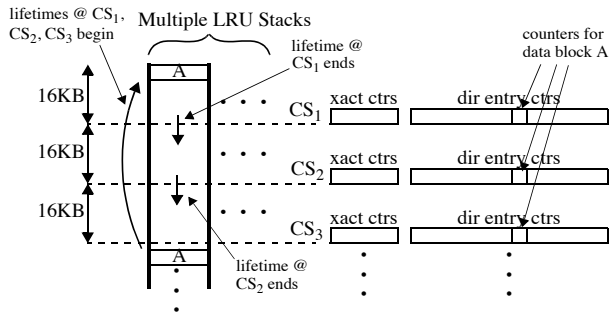

Figure 5: Counters implemented in the PIN profiler.

to compute PRD and PRD_{remote}, using Table 1 to determine the cache transaction and directory lookup type. Most T1/T2 transactions perform read-modify-writes, but our profiler attributes a single directory cache access to each transaction. Section 5 will break down the reads and writes separately.

To enable capacity-sensitive analysis, our PIN profiler refers to Table 1 multiple times per memory reference, determining the directory cache 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 profiling speed. For each *CS* value, we maintain 18 counters, one per cache transaction in Table 1, and increment the corresponding counter based on the result from the table. Figure 5 illustrates the per-transaction counters at each profiled private cache size (labeled "xact ctrs").

In addition to counting cache transactions, our PIN profiler also counts lookups to individual directory entries during their lifetimes in the directory cache. We maintain a set of directory entry counters, *one per unique data block contained in all of the LRU stacks*, at every profiled capacity. Figure 5 illustrates these counters (labeled "dir entry ctrs"). After updating the "xact ctr" at a particular *CS*, we also check if the transaction causes a directory lookup. If so, we increment the corresponding "dir entry ctr" to register the directory entry's access at that *CS* value. Whenever the copies of a data block are pushed below a certain size CS_i across all LRU stacks, the data block's lifetime in the private caches ends. Our profiler assumes the directory entry's lifetime in the directory cache also ends (a simplification we will address in Section 5). Moreover, the corresponding "dir entry ctr" reflects the number of lookups the directory entry received during its lifetime given private caches of size CS_i. We record this counter value in a histogram for CS_i, and clear it to prepare for the next lifetime. Figure 5 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. (This example assumes block *A* is at depth $> CS_3$ in all other LRU stacks).

Finally, our PIN profiler follows McCurdy's method [19] 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 [16, 28], especially for PRD profiles.

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

Table 2: Parallel benchmarks used in the evaluations.

We profiled 15 parallel benchmarks. Our results assume 64 threads running on 64 cores (*i.e.* 64 LRU stacks). Table 2 lists the benchmarks and their suites: SPLASH2 [26], MineBench [21], or PARSEC [6]. The last two columns in Table 2 report the problem sizes and their dynamic instruction counts (in billions). For the kernels (indicated in the first column), we profiled the entire benchmark run. For all other benchmarks, we ran the first parallel iteration to warm up the PRD stacks, and then profiled the second parallel iteration.

4.2. Access Frequency

As described in Section 2, the directory cache is looked up on data cache misses (T1 + T2). Table 3 and Figure 6 present our cache miss results for our benchmarks running on 64 cores. In particular, the middle 3 columns of Table 3 report private cache miss rates, and hence, the directory cache's access rate, as measured by our PIN profiler. Results are shown for private cache sizes of 1KB, 256KB, and 1MB per core. At 1KB, the directory cache lookup rate is between 4–35%. But for more realistic cache sizes, lookups drop quickly. By 256KB, on average, only 1.7% of all memory references lookup the directory cache, and by 1MB, only 1.3%.

The solid lines in Figure 6 labeled "Total Lookups" plot the private cache misses (or directory cache lookup accesses) per 1000 instructions–*i.e.* MPKI or "APKI"–as a function of private cache size. The X-axis spans a wide range of cache sizes, from 16KB to the maximum PRD observed in each benchmark. Like Table 3, these results also show lookups drop to low values as cache capacity scales. Averaged across all benchmarks, the directory cache experiences 5.6 and 4.0 APKI at private cache sizes of 256KB and 1MB, respectively.

Granted, cache miss behavior is highly application and problem size dependent. For our benchmarks, Table 3 and Figure 6 show directory cache lookups are fairly infrequent. But in general, one can expect a significant fraction of memory references to be filtered from the directory cache.

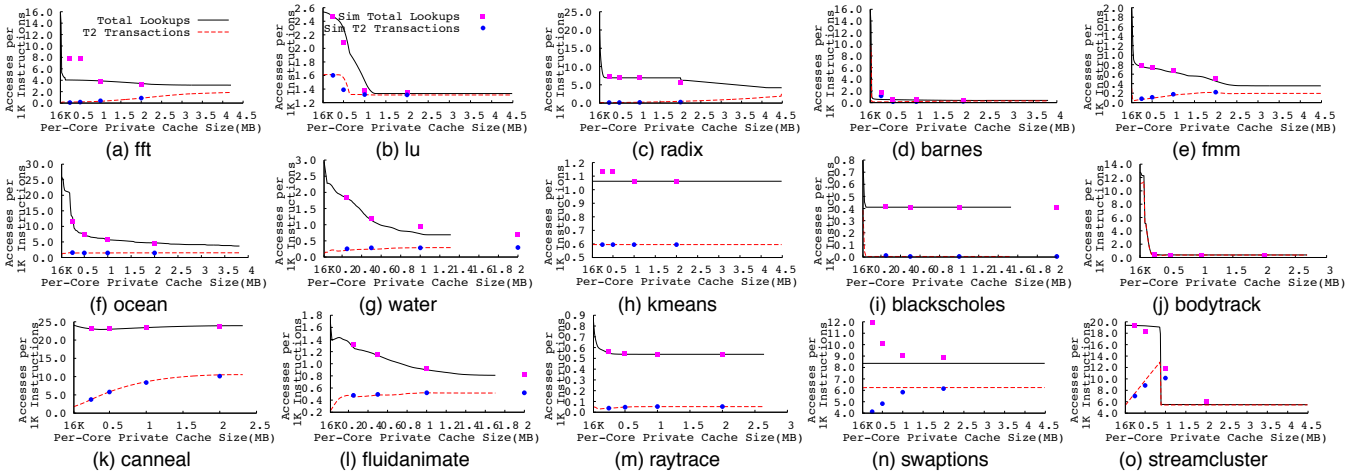

Figure 6: Number of total and T2-type directory cache lookups per 1000 instructions as a function of private cache size.

Benchmark	Dir Access Rate			T2 Coverage	
	1KB	256KB	1MB	256KB	2MB
fft	9.5%	1.7%	1.7%	10.0%	28.3%
lu	6.8%	0.7%	0.4%	99.9%	100.0%
radix	21.9%	3.1%	3.1%	92.9%	28.8%
barnes	4.0%	0.1%	0.1%	90.2%	95.2%
fmm	34.4%	0.5%	0.4%	72.4%	94.7%
ocean	14.5%	3.3%	1.5%	95.3%	98.1%
water	11.5%	0.4%	0.2%	60.0%	83.1%
kmeans	4.3%	0.4%	0.4%	100.0%	100.0%
blackscholes	10.6%	0.1%	0.1%	100.0%	100.0%
bodytrack	8.1%	0.1%	0.1%	97.7%	99.9%
canneal	20.6%	7.1%	7.2%	65.7%	93.3%
fluidanimate	8.3%	0.3%	0.2%	91.1%	99.6%
raytrace	4.0%	0.1%	0.1%	61.2%	44.6%
swaptions	12.4%	2.2%	2.2%	96.1%	96.1%
streamcluster	7.2%	5.7%	1.6%	70.4%	100.0%
Average	11.9%	1.7%	1.3%	80.2%	84.1%

Table 3: Directory cache access rates and T2 transaction coverage in ≥ 3 lifetimes for different private data cache sizes.

Next, we study how accesses are distributed across individual directory entries. Figure 7 breaks down all directory entry lifetimes observed throughout our benchmarks' runs in terms of number of lookups received at different private cache sizes. In each graph, five curves plot the fraction of directory entries receiving ≥ 1, ≥ 2, ≥ 3, ≥ 5, and ≥ 10 lookups while resident in the directory cache. These breakdowns are shown across the same range of private cache sizes as in Figure 6.

Figure 7 shows for smaller private caches, the vast majority of directory entries receive very few lookups during their lifetimes. For example, at 256KB private caches, 93% of all directory entry lifetimes experience only 1 or 2 lookups (*i.e.* the gap between the "1" and "3" curves) averaged across all benchmarks. In fact, most directory entry lifetimes experience only a single lookup: 86% on average.

The *singleton lifetimes* are due to private data. As mentioned earlier, data blocks referenced by a single core do not generate directory lookups while resident in cache. For such private blocks, our profiler observes one directory lookup for the initial data cache fill–a T1 transaction–but does not observe any further lookups (*i.e.* there are no T2 transactions that reuse the filled entry). So, for 256KB private caches, not only are directory cache accesses infrequent, as shown in Table 3 and Figure 6, but the majority of directory entries within the directory cache exhibit very low reuse due to private data.

For larger private caches, however, many of our benchmarks exhibit increased sharing and more lookups per directory entry. As discussed in Section 3.2, sharing may occur between threads but remain invisible to on-chip coherence mechanisms due to insufficient capacity to capture the sharers. Indeed, Figure 7 shows as private cache size scales, the fraction of directory entry lifetimes that experience reuse (*i.e.* ≥ 2 lookups) increases in half the benchmarks. For fft, lu, barnes, bodytrack, and streamcluster, the private cases that dominated at 256KB are in the minority by 2MB, making up only 21% of all directory entry lifetimes. A similar trend occurs for radix, fmm, and canneal, though the private cases still dominate at 2MB in those benchmarks. These results show there are many temporally private blocks at 256KB which become shared by 2MB, exposing more T2 transactions to the directory cache.

Overall, this redistribution of private *vs.* shared blocks with capacity scaling reduces the number of low-reuse directory entries in the directory cache, but low-reuse is still the predominant behavior. Averaged across all benchmarks, Figure 7 shows 69% of directory entry lifetimes experience only 1 or 2 lookups given 2MB of private cache (instead of 93% at 256KB), with 62% experiencing only a single lookup.

4.3. Latency Tolerance and Temporal Locality

As described in Section 2, directory cache accesses for T2 transactions are latency critical whereas those for T1 transactions are latency tolerant. Figure 6 quantifies the T2 *vs.* T1 balance. In addition to the solid lines plotting total directory lookups, Figure 6 also plots as dotted lines the lookups associated with T2 transactions only. Hence, the gap between the solid and dotted lines quantify the T1 lookups.

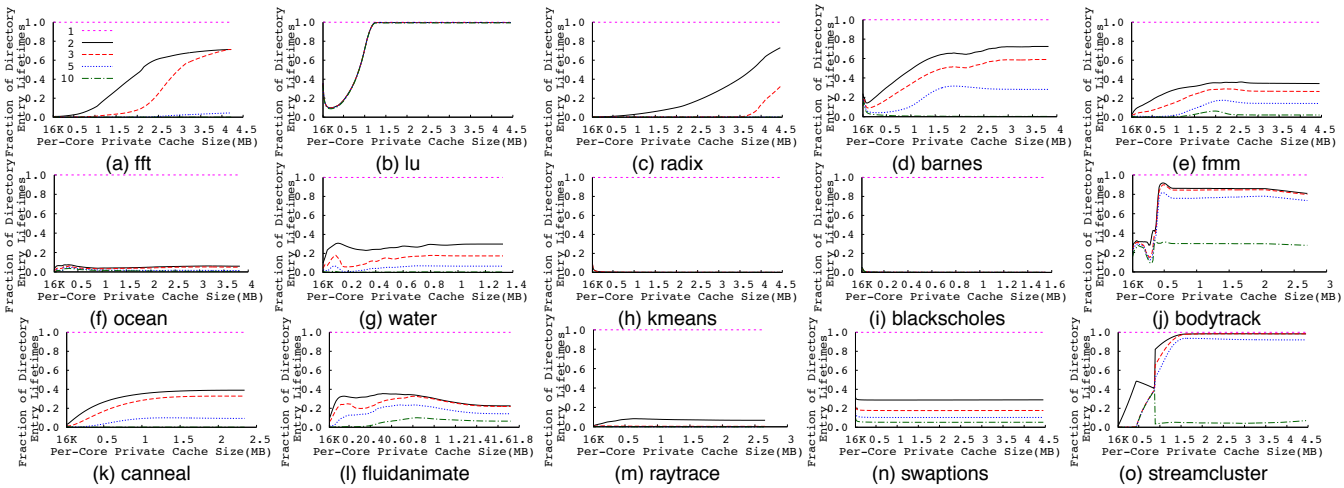

Figure 7: Distribution of lookups over individual directory entries during their lifetimes as a function of private cache size.

Figure 6 shows a large fraction of directory cache lookups are latency tolerant, especially at small private caches. Given 256KB private caches, 71.1% of directory lookups on average are T1 transactions. These results are consistent with our per-entry analysis. In particular, the singleton lifetimes for private data in Figure 7 each generate a T1 transaction, and as discussed in Section 4.2, these dominate at 256KB. Many of the non-singleton lifetimes in Figure 7 contribute multiple T2 transactions and reduce the fraction of T1 transactions. Still, most directory cache lookups are latency tolerant at 256KB.

For larger private caches, a significant fraction of directory lookups become latency critical. At 2MB private caches, about half (50.8% on average) are latency critical and half are latency tolerant. This is mainly due to the drop in cache misses with scaling as discussed in Section 4.2, eliminating cache fills and their T1 transactions. Interestingly, Figure 6 shows T2 transactions also drop in some cases (lu, barnes, blackscholes, bodytrack, and streamcluster). This is due to read sharing. As cache size grows, forwarding transactions (#11 in Table 1) increase as remote sharers are captured on-chip. But once *all sharers* are cached, the directory lookups are eliminated–*i.e.* the read-sharing working set fits in cache.

In contrast, write sharing causes coherence-related misses (#9, 10, 12, and 13 in Table 1). These also increase with capacity scaling, but unlike read sharing, they cannot be eliminated by capturing the sharing on-chip. This causes the steady rise in T2 transactions visible in Figure 6, which increases overall latency sensitivity. Notice, at each benchmark's maximum PRD, all read-shared T2 transactions are eliminated but all write sharing becomes exposed. These "∞" private caches quantify the maximum T2 transactions possible from write-sharing. Figure 6 shows such worst-case write sharing generates only 2.1 lookup APKI on average. For small private caches, the T1 transactions easily dominate this write sharing traffic. So, unless read sharing is significant, directory caches can be highly latency tolerant given small private cache sizes.

Finally, because T2 transactions are concentrated in non-singleton directory entry lifetimes and because these are in the minority, there is significant temporal reuse of directory entries across T2 transactions. The last two columns in Table 3 report the percentage of T2 transactions that access directory entries with ≥ 3 per-lifetime lookups. Results are shown for 256KB and 2MB private caches. Although ≥ 3 lifetimes only make up 7% of all directory entry lifetimes at 256KB (see Section 4.2), Table 3 shows they account for 80.2% of all T2 transactions on average. A similar T2 transaction coverage is achieved at 2MB, 84.1%, though more directory entry lifetimes are involved, 31%. These results show a small directory cache can potentially hold the majority of directory entries accessed by latency-critical T2 transactions.

5. Cache Simulation Experiments

This section evaluates the caching performance of two-level directory caches. We first discuss our simulator infrastructure. Then, we present the results without and with shared caches.

5.1. Directory Cache Simulator

We implemented a cache simulator that models two versions of the hierarchy in Figure 1. Our simulator uses the same PIN tool from Section 4.1 except the LRU stacks are replaced by cache models. In one hierarchy, we use three levels of private cache, and in the other, we use two levels of private cache and a shared cache. All caches are inclusive. For both hierarchies, we maintain private cache coherence via a directory-based MESI protocol with full-map directories. (While other implementations are possible, full-map eliminates transactions for imprecise sharer encodings, leaving the T1 and T2 accesses our study focuses on). To show scaling effects, we simulate four different last-level private cache sizes in the private-only hierarchy. Table 4 lists the simulation parameters.

Our simulator also models a two-level directory cache. We choose parameters (see below) that are suited for Figures 2B and C, but our cache performance and traffic results provide insights into all options in Figure 2. Besides the data cache misses our profiler analyzes, our simulator's directory cache also receives writes from data cache evictions. (Our MESI

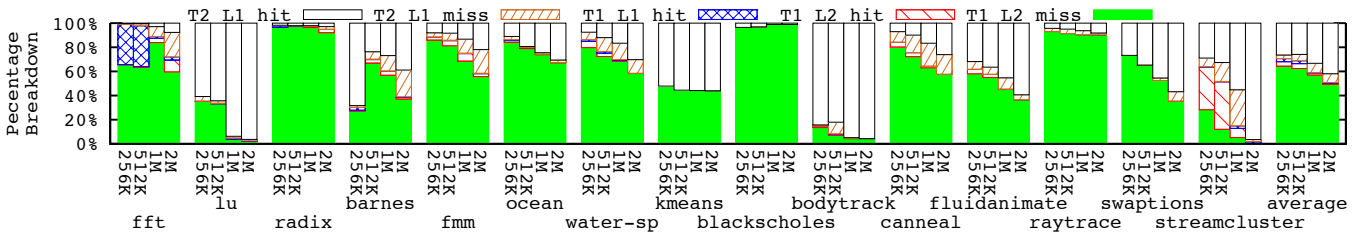

Figure 8: Hit and miss rates for T1 and T2 lookups at different levels of the directory cache for different private cache sizes.

Data Cache Hierarchy	
Private Only	Private + Shared
Private L1: 16 KB, 4-way Private L2: 64KB, 8-way Private L3: 256KB, 512KB, 1MB, or 2MB, 8-way	Private L1: 16 KB, 4-way Private L2: 256KB, 8-way Shared L3: 64MB, 8-way
Directory Cache Hierarchy	
L1: 732KB, 1.4MB, 2.8MB or 5.7MB, 6-way L2: 30.5MB, 61MB, 122MB, or 244MB, 1-way	

Table 4: Cache simulation parameters. All data cache blocks are 64 bytes.

protocol notifies the directory when evicting modified and exclusive data blocks [13], while shared data blocks are silently evicted). Another difference is that in our simulator, a directory entry's lifetime ends when it is evicted from the directory cache, not when its data block(s) are evicted from the data caches. This results in longer lifetimes.

Two-level directory cache management is as follows. On a directory lookup, the L1 is checked first, and then the L2 is checked if the L1 misses. On an L2 miss (a T1 transaction), a new directory entry is filled into the L1 (a new lifetime) where it may receive 1 or more hits (fast T2 transactions). Eventually, the entry is evicted from the L1 and gets filled into the L2 via a writeback. On an L2 hit (a slow T2 transaction), the directory entry is promoted to the L1, leaving its copy in the L2. L1 entries that were inserted via T2-based promotions may not be modified, so the L1 maintains per-entry dirty bits to avoid unnecessary L2 writebacks. Both the L1 and L2 are set associative, performing evictions in per-set LRU order.

The L1 and L2 directory caches are sized to be a fixed fraction of the total last-level private data cache. Table 4 reports four L1 and L2 directory cache sizes, one for each of the four last-level private cache sizes we simulated. The L1 and L2 directories are always 4.5% and 91%, respectively, of the total private data cache. (In terms of the "directory entry to data cache block ratio," the L1 and L2 provide 0.18X and 8X "coverage," respectively, of the private caches). The L1 sizes were chosen to match the average footprint for frequently accessed directory entries (see Section 4.3). The L2 sizes were chosen to minimize conflicts. Given its potentially high access cost (*e.g.* Figures 2B/C), we made the L2 direct mapped so each L1 miss only performs one L2 access. We found 8X coverage mitigates the L2 conflicts. While the L2 sizes are large, they are still small compared to what DRAM can provide.

5.2. Private Cache Results

We begin by comparing the directory access counts from our simulator against those in Section 4 to help validate our analysis framework. Figure 6 plots the APKI measured on our simulator for both total and T2-type directory cache lookups, labeled "Sim Total Lookups" and "Sim T2 Transactions," respectively. There are four simulation datapoints per APKI measurement, one for each of the four private cache sizes we consider. In water and fluidanimate, the 2MB cache size is larger than the maximum PRD for those benchmarks, so these simulation datapoints should be compared against the corresponding access counts at the benchmarks' maximum PRD.

As Figure 6 shows, the simulator and profiler results are very close in most cases. Averaged across all simulation points and benchmarks, the error in total APKI is only 9.2% while the error in T2-type APKI is 13.6%. For total APKI, the simulator is usually higher than the profiler. This is because our analysis framework uses ideal LRU stacks which do not model cache conflicts, thus underpredicting the actual cache misses that lookup the directory. In contrast, cache conflicts do not systematically affect the amount of on-chip sharing. So for T2-type APKI, the simulated results can be higher or lower than the analysis results; however, there is no bias.

Next, we look at how effectively multi-level directory caches exploit temporal locality. Figure 8 breaks down the rate at which T1 and T2 lookups hit in different directory caches as a percentage of total accesses. For each benchmark, groups of bars report the results for different private cache sizes, with each bar breaking down five categories: T2 accesses that hit or miss in the L1 directory cache, T1 accesses that hit in either the L1 or L2 directory caches, and T1 accesses that miss in both directory caches.

Figure 8 shows the majority of T2 lookups hit in the L1 directory cache. As the "T2 L1 Hit" and "T2 L1 Miss" components show, the T2 hit rate across different cache sizes is between 26–42% for all lookups, which is 75–80% for the T2 lookups. Recall from Section 4.3 that 7–31% of frequently accessed directory entries account for roughly 80–84% of all T2 lookups. Our simulations confirm a very small L1 directory cache can exploit this temporal reuse. Hence, most latency-critical T2 lookups are serviced rapidly, so only a minority–about 20% on average–incur longer L2 accesses.

Figure 8 also shows the majority of lookups miss in both directory caches. These lookups, labeled "T1 L2 Miss," are the T1 directory fills that initiate new directory entry lifetimes. As Figure 8 shows, they account for 49–64% of all directory

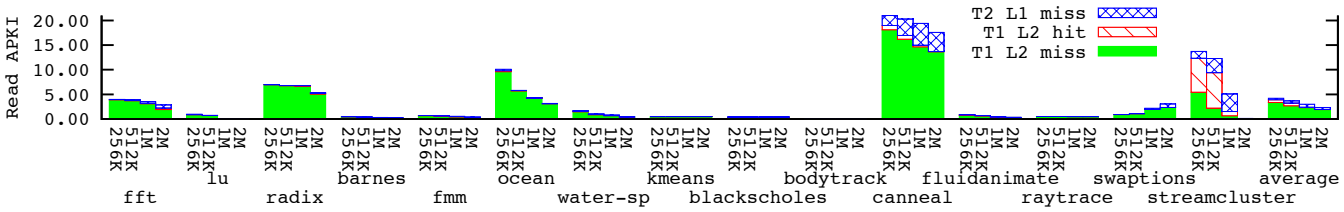

Figure 9: L2 directory cache read APKI for different private cache sizes.

lookups across different cache sizes, which is consistent with our analysis framework (see T1 *vs*. T2 validation above). Notice, Figure 8 also shows some T1 lookups *hit* in the L1 or L2 directory caches. Directory entries may linger in the directory cache after their data blocks are evicted, an effect our analysis framework does not model. During these extended lifetimes, an entry can receive more accesses which may be significant for small caches that repeatedly evict frequently accessed blocks (fft and swaptions). But Figure 8 shows only 2.9–10.2% of T1 lookups hit in the L1 or L2 on average. So, our analysis framework accounts for most T1 lookups.

Lastly, Figure 8 shows T1 speculation will be highly accurate. On an L1 directory cache miss, if we speculate the transaction is of type T1 (see Section 2.1), then we will be wrong only 12% of the time on average (*i.e*. the percentage of L1 misses that are of type T2). In these mispeculated T2 cases, we will needlessly fetch a data block from main memory, but for all other L1 misses, we will gainfully overlap the L2 directory access with its data access.

Figures 9 and 10 report traffic from/to the L2 directory cache. The former shows read traffic due to lookups in Figure 8 that could not be serviced from the L1 ("T2 L1 miss," "T1 L2 hit," and "T1 L2 miss" components) and the latter shows write traffic due to writebacks for both dirty L1 directory entries and private data blocks in modified or exclusive state. We use the same format as Figure 8, except APKI is plotted along the Y-axis.

Figures 9 and 10 show L2 directory cache traffic is application dependent. For 9 benchmarks, the traffic is very small, with total read and write traffic always under 2 APKI. This is due to low directory lookup rates reported in Section 4.2 and filtering of many T2 lookups by the L1 directory cache. But in the remaining benchmarks, the traffic can be significant. For fft, radix, ocean, swaptions, and streamcluster, read and write traffic are each between 4.1–21.0 APKI, while in canneal, they can each reach 25.0–34.2 APKI. Across all benchmarks, read and write traffic are each between 2.3–6.5 APKI.

While the L2 directory cache traffic can be significant in some cases, it is almost entirely off the CPU's critical path. The vast majority of read traffic consists of latency tolerant T1 lookups. Moreover, *all* of the write traffic in Figure 10 can be buffered and performed without stalling the CPU. Although writes are significant in canneal and a few others, conventional DRAM should be able to keep up with these write APKI, especially given the writebacks can be pipelined. In contrast, except for canneal and streamcluster, Figure 9 shows latency critical T2 lookups are very rare. Across all benchmarks, they comprise 13% of all reads.

Benchmark	No Shared	Shared	Shared+Deferred
fft	0.07	0.02	0.02
lu	0.10	0.10	0.01
radix	0.02	0.04	0.04
barnes	0.02	0.04	0.01
fmm	0.03	0.04	0.03
ocean	0.33	0.49	0.30
water	0.12	0.21	0.08
kmeans	0.00	0.00	0.00
blackscholes	0.00	0.00	0.00
bodytrack	0.01	0.01	0.01
canneal	2.04	2.90	1.42
fluidanimate	0.09	0.14	0.07
raytrace	0.02	0.02	0.01
swaptions	0.08	0.05	0.04
streamcluster	1.38	8.27	0.04
Average	0.29	0.82	0.14

Table 5: Latency critical L2 directory cache read APKI for 256KB private caches with or without shared caches and deferred updates.

5.3. Shared Cache Impact

As discussed in Section 2, directory cache behavior is mainly determined by private caches, but having a shared cache can make some T1 transactions latency critical. While still absent from the private caches, the data for certain T1 transactions may now be found in the on-chip shared cache, eliminating their off-chip data accesses and potentially putting their slower L2 directory accesses on the CPU's critical path.

Table 5 shows how many L2 directory cache accesses become latency critical with a shared cache. The second column in Table 5 reports the read APKI for latency critical transactions in a private-only hierarchy with 256KB private caches (*i.e*. the "T2 L1 miss" components from the "256K" bars in Figure 9). Then, the third column reports the same but for a hierarchy that includes a 64MB shared cache in which shared cache hits for T1 lookups are treated as latency critical.

Averaged across all benchmarks, Table 5 shows the latency critical portion of the L2 directory's read traffic increases from 0.29 APKI without a shared cache to 0.82 APKI with a shared cache. This is due to the "T1 L2 hit" components in Figure 9. These L2 accesses correspond to data that were recently evicted from the private data caches, and hence, their directory entries are still in the L2 directory cache. They mostly involve data blocks that were in shared state whose evictions do not notify the directory. Because of good temporal reuse, there is a high likelihood for these accesses to hit in the shared cache. Our results show essentially all of the "T1 L2 hit" components move into the latency critical category.

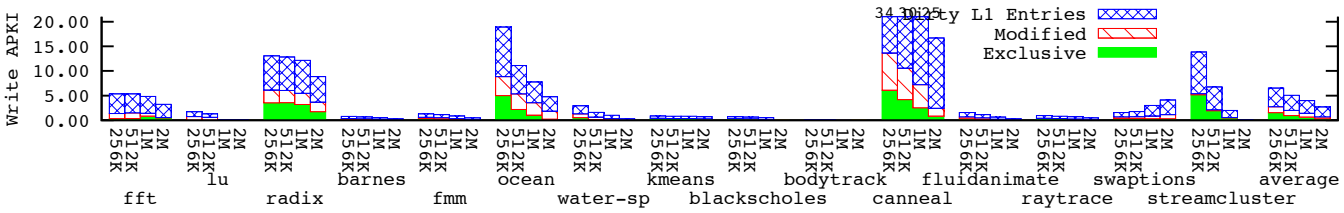

Figure 10: L2 directory cache write APKI for different private cache sizes.

While shared cache hits eliminate one source of latency tolerance, it may still be possible to overlap a T1 transaction's L2 directory access if we *defer the directory entry update*. For many shared cache hits, the T1 transactions will find the data blocks in shared state as mentioned above. If the T1 access is a data read and the shared cache includes the state information as part of its cache tags, then the shared cache data can be forwarded to the requesting core immediately without further stalling the core. Granted, the L2 directory access must still occur (*i.e.* we need to add a sharer to the sharing vector), but this can happen off the critical path of the core's data access.

The last column in Table 5 shows the impact of this optimization: the latency critical read APKI drops to 0.14 on average, becoming even lower than the original traffic without a shared cache. The reason is because we can also defer directory entry updates for some T2 transactions performing read forwarding (*i.e.* #11 in Table 1). In these cases, the requesting core does not need to wait for the directory access to determine the remote sharer from which to forward since the data is already available in the shared cache. So, deferred updates address latency criticality for T1 transactions *and* make certain T2 transactions latency tolerant as well.

6. Related Work

Recently, there has been significant interest in developing RD analysis techniques for parallel programs by adding models for thread interactions. Our work exploits PRD profiling [24, 23, 28] which accounts for thread interactions in private caches. There has also been work for shared caches [12, 16, 27, 28]. But all prior art has focused on analyzing data reuse. Our work further extends RD analysis to reason about reuse across the directory state.

A significant amount of research has been devoted to minimizing the directory's size. Sparse directories [14] only keep directory entries for data blocks resident in the private data caches, but require significant over-provisioning to reduce conflicts. Limited pointers [1, 5, 7, 8, 9] and coarse vectors [14] reduce sharing vectors at the expense of extra coherence messages. Tagless directories [30] use bloom filters to compact the directory. Cuckoo [13] employs sophisticated hashing schemes to reduce conflicts without significant over-provisioning. And SCD [22] uses variable directory tags to match the sharing vector width to actual sharing set size.

Our approach is orthogonal to these existing techniques. Rather than compact the directory, we exploit variation in access frequency and latency tolerance to split the directory into smaller/faster *vs.* larger/slower caches. Although we evaluate a very simple full-map scheme, this general approach could be applied to *any* of the above techniques given the Figure 2A option (except perhaps Tagless directories). The DRAM options in Figures 2B and C, which are better matched to the simulation parameters in Section 5, are more limited. The above techniques could be applied to the L1 directory cache, but given its high access cost, only simple techniques like limited pointers could be applied to the DRAM-based L2 directory cache. But our research provides the analyses for deeply understanding all of these multi-level architectures.

Our work is closely related to PS-Dir [25] and WayPoint [17], early implementations of two-level directory caches. PS-Dir and other work by the same authors [11, 10] are the first to observe the importance of privately accessed data. And, WayPoint observes there is temporal reuse on certain directory entries. Compared to these papers, our work focuses on analyzing the directory access patterns, and providing deep insights into those access patterns. In particular, we agree private data is important, but our results show sharing can become dominant in some cases, especially for large caches. Our analyses quantify how private *vs.* shared entries vary across data cache size. Similarly, we quantify the continuum of temporal locality that the WayPoint paper talks about qualitatively. We also show the contribution of write sharing to directory entry reuse, which is much less significant in the WayPoint work. (They focus on streaming workloads that do not exhibit much write sharing). Lastly, we also quantify latency tolerance which is not considered by either work.

Multi-level directory caches have also been studied in the context of CC-NUMA machines [2, 3, 4, 15, 20]. In this case, the directory is far away from the cores, so directory entries are cached *above the sharing point* near the cores to reduce access latency. In multicores, the directory and sharing point are close to the cores but storage is limited, so directory entries are cached *further below the sharing point* to improve scalability. Conceptually, the two are very similar. We focus on analyzing directory access patterns for multicores. However, our analysis framework could be modified to reason about CC-NUMA directory caches.

7. Conclusion

This paper develops RD-based techniques for analyzing directory cache access patterns. Our analysis framework can quantify directory cache accesses and their distribution over individual entries, as well as break down the number of latency tolerant *vs.* critical accesses. Also, our framework can perform these analyses for all private cache sizes. Using these

techniques, we show most directory entry lifetimes, 69–93%, receive only 1 or 2 lookups, with the majority of latency critical reuse accesses confined to a small number of directory entries. We also find a majority of directory cache accesses, 51–71%, are latency tolerant. Our work also conducts cache simulations to quantify the caching performance of multi-level directory caches. We find very small L1 directory caches (with only 0.18X coverage) can service 80% of latency critical directory accesses. Although a significant number of cache transactions access the slower L2 directory cache, virtually all of these are latency tolerant.

References

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

[2] M. E. Acacio, J. González, J. M. García, and J. Duato, "A novel approach to reduce l2 miss latency in shared-memory multiprocessors," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, ser. IPDPS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 25–.

[3] ——, "Reducing the latency of l2 misses in shared-memory multiprocessors through on-chip directory integration," in *Proceedings of the 10th Euromicro conference on Parallel, distributed and network-based processing*, ser. EUROMICRO-PDP'02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 368–375.

[4] M. Acacio, J. Gonzalez, J. Garcia, and J. Duato, "An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 8, pp. 755–768, 2004.

[5] 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, 1988.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 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, 1991.

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

[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, D.C., 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, pp. 482–495, 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: ACM, 2011, pp. 93–104.

[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] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 169–180.

[14] A. Gupta, W. dietrich Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *In International Conference on Parallel Processing*, 1990, pp. 312–321.

[15] R. Iyer, L. Bhuyan, and A. Nanda, "Using switch directories to speed up cache-to-cache transfers in cc-numa multiprocessors," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, 2000, pp. 721–728.

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

[17] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "Waypoint: Scaling coherence to thousand-core architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 99–110.

[18] 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*, 2005.

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

[20] M. M. Michael and A. K. Nanda, "Design and performance of directory caches for scalable shared memory multiprocessors," in *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, ser. HPCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 142–.

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

[22] 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*, 2012.

[23] 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*, 2010.

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

[25] 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*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 451–452.

[26] 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*, 1995.

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

[28] ——, "Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs," *ACM Transactions on Computer Systems*, vol. 31, no. 1, 2013.

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

[30] 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, 2009.