# Stash Directory: A Scalable Directory for Many-Core Coherence

Socrates Demetriades[†] and Sangyeun Cho[‡†]
[†]Computer Science Department, University of Pittsburgh
[‡]Memory Division, Samsung Electronics Co.
{socrates,cho}@cs.pitt.edu

## Abstract

*Maintaining coherence in large-scale chip multiprocessors (CMPs) embodies tremendous design trade-offs in meeting the area, energy and performance requirements. Sparse directory organizations represent the most energy-efficient and scalable approach towards many-core coherence. However, their limited associativity disallows the one-to-one correspondence of directory entries to cached blocks, rendering them inadequate in tracking all cached blocks. Unless the directory storage is generously over-provisioned, conflicts will force frequent invalidations of cached blocks, severely jeopardizing the system performance. As the chip area and power become increasingly precious with the growing core count, over-provisioning the directory storage becomes unsustainably costly.*

*Stash Directory is a novel sparse directory design that allows directory entries tracking private blocks to be safely evicted without invalidating the corresponding cached blocks. By doing so, it improves system performance and increases the effective directory capacity, enabling significantly smaller directory designs. To ensure correct coherence under the new relaxed inclusion property, stash directory delegates to the last level cache the responsibility to discover hidden cached blocks when necessary, without however raising significant overhead concerns. Simulations on a 16-core CMP model show that Stash Directory can reduce space requirements to* $1/8$ *of a conventional sparse directory, without compromising performance.*

## 1 Introduction

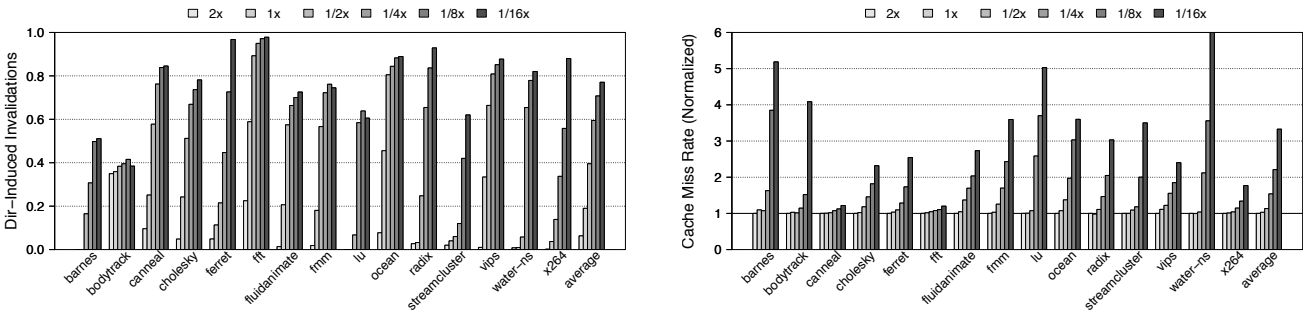
As the number of cores grows in chip multiprocessors (CMPs), maintaining coherence across private caches becomes increasingly expensive. To meet the scalability challenges, today's coherence protocols employ directories to precisely track and control the multiple copies of data that may exist in different locations on chip. While the advances in semiconductor technology are expected to continue the exponential growth in core count, there is a serious concern that employing directory coherence (and coherence in general) will quickly become problematic due to high storage and energy requirements. Barring the concerns, Martin et al. [20] argue that "cache coherence is here to stay" as its strong legacy and transparent performance are hard to ignore. Overcoming the scalability limitations of directory coherence is therefore becoming essential in designing future many-core CMPs.

Directory organizations embodied in modern CMPs make different trade-offs in meeting the area, energy and performance requirements. Duplicate-tag directories [3, 21] are simple to implement and incur low area cost; however, they require associativity proportional to the core count, which makes them extremely energy-inefficient even for systems with 4 to 8 cores. "In-Cache" directories take advantage of chip architectures with a shared last level cache (LLC) and organize the directory information as part of its entries [25]. This approach saves storage and energy since no extra tags or lookups are necessary. However, each directory entry must encode a set of sharers, which grows linearly with the number of cores. Furthermore, the sharer set must be kept for every LLC tag, independent of whether the associated block is cached or not. While clever sharer encoding methods alleviate the former scalability limitation [12, 26], the redundantly allocated storage makes them area-inefficient for future many-core systems.

Alternatively, sparse directories [12] offer separate address-indexed arrays to flexibly cache directory entries, decoupled from the LLC design. This preserves the energy benefits of low associativity and enables more space efficient designs. Their limited size and associativity, however, disallows the one-to-one correspondence of directory entries to cached blocks, preventing them from being able to track all possible cached blocks, and forcing them to invalidate cached blocks when conflicts occur. Unless the storage of sparse directories is over-provisioned, conflicts could occur frequently and severely hurt the system performance.

Prior research highlights the need for more effective use of a sparse directory's available space as a way to reduce the over-provisioning requirements [10, 11, 23]. For example, Cuckoo [11] and SCD [23] directories explore multi-hashing to resolve set conflicts and reduce directory-induced invalidation. These techniques achieve better space utilization and provide theoretical bounds; however, they are limited in potential as they are general to any set-associative structure and they do not exploit any character-

978-1-4799-3097-5/14/$31.00 




(a) Directory-Induced Invalidations

(b) Cache Miss Rate

Figure 1: *Directory-Induced Invalidations (a) and their effect on Cache Miss Rate (b) as a Function of the Directory Provisioning Ratio:* The rate in (a) is the number of invalidation ops to the total number of directory accesses. Miss rate is normalized to a $2\times$ Sparse Directory.

istics germane to the directory behavior. In a more recent study, Cuesta et al. [10] increase the effectiveness of the directory space by deactivating coherence for private data. While this approach is better in exploiting the unique properties of the directory, it relies on the TLB miss handler and the page table to detect private data, a granularity that is too large to grasp the full potential. Lastly, Alisafaee [2] presents a directory which discovers and compacts temporarily private data to save space. Although this approach is effective for programs with large, highly shared data footprints, its usage and complexity are hard to be justified for the general case.

Instead of focusing on reducing directory conflicts, this work aims to relax required functionality of sparse directories to enforce invalidation on every eviction. The key observation is that most of the evictions in directory force invalidation of private blocks—a requirement that is not only unnecessary from the coherence perspective, but also severely harmful to the cache performance. By tolerating a large number of directory evictions without forcing invalidation, we can eliminate the need for storage overprovisioning while retaining or improving the effectiveness of the cache, and in turn, the overall system performance.

This paper proposes *Stash Directory*, a novel sparse directory design that aggressively avoids any induced invalidations of blocks that appear to be private to processors. By doing so, it transparently eliminates performance loss due to premature directory-induced invalidations of blocks in the private caches, and at the same time reduces pollution in the directory caused by subsequent recalls of those blocks—effectively improving the use of the available directory space. As private blocks are often dominant in parallel and multiprogramming workloads [10], Stash Directory is scalable with nearly constant power and area utilization regardless of the core count.

Since Stash Directory allows cached blocks to be left intractable, or "hidden" from the the directory, the coherence protocol must ensure that no coherence inconsistencies can occur. This is achieved by delegating to the LLC the responsibility of keeping a "cached" status bit for each block. The coherence protocol leverages this information to detect whether a miss in the directory refers to a block that is not owned exclusively by the LLC, but is in fact cached in some private cache. In such case, the protocol falls back to broadcasting to find and retrieve the most up-to-date copy.

Experimental results using parallel workloads running on a 16-core CMP show that Stash Directory can use an area as small as $1/8$ that of a conventional over-provisioned sparse directory, with no performance loss or significant additional bandwidth requirements. Compared to a conventional sparse directory of an equivalent size, Stash Directory can improve performance by 16% and cache energy consumption by 18%. Analytical models show that inconsistent misses are rare events that are approximately bounded by application-specific characteristics and machine parameters. Overall, Stash Directory is shown to scale exceptionally well with the core count, offering an attractive solution among the leading directory organizations for large-scale CMPs.

In the remainder of this paper, we will first discuss the background and the motivation of this work in Section 2, and then describe Stash Directory in Section 3. Section 4 gives the methodology of our evaluation, followed by the evaluation results in Section 5. Section 6 will discuss the related work to this work. Finally, Section 7 will conclude.

## 2 Background and Motivation

### 2.1 Directory-Induced Invalidation

Sparse directories [1, 6–8, 10–12, 22–24, 26, 29] are organized as an associative array indexed by block address, and each directory tag encodes the set of sharers of the associated block. Sparse directories are energy-efficient; however, due to their limited associativity, they are subject to set conflicts that result in evictions of set entries. To preserve the correctness of the cache coherence, sparse directories force invalidation of the cached blocks that correspond to the evicted entries. This can pose large performance overheads due to additional cache misses in processors' private caches.

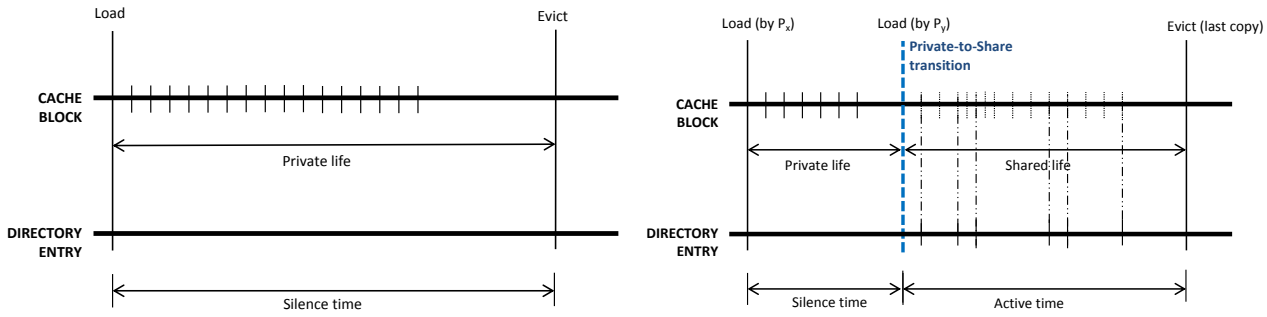



Figure 2: *Lifetime Behavior of Cache Block and Directory Entry:* The life of a private block (left) vs. a shared block (right).

Figure 1a and 1b show, respectively, the rate in which the directory forces invalidations back to the private cache hierarchy and the corresponding increase in cache miss rate, for different sizes of a sparse directory. The different sizes are labeled as *ratios*, which denote the provisioned size of the directory. For example, a $1\times$ corresponds to a system with as many directory entries as the number of blocks that can be cached. The invalidation rate (and in turn the increase in miss rate) mainly depends on the relative directory and cache size and associativity, and the working set size of a given application. The smaller the directory, the more set conflicts, and consequently the more directory-induced invalidations, will occur. If the working set of the application fits in the directory, no conflicts occur and performance is ideal.

Reducing the eviction frequency, and therefore the resulting extra misses, requires over-provisioning the number of directory sets and associativity. As the results show, and as previous research and real implementations have also suggested [9, 12], set-associative directories need to be over-provisioned by more than $2\times$ in order to reduce evictions, while still providing no guarantees. CMP architectures that implement a directory as part of the shared LLC, i.e., in-cache, can achieve a near ideal case since they naturally encompassed an over-provisioned design. However, such implementations tend to result in an extremely over-provisioned case since the number of tags in the LLC could greatly exceed the number of tracked blocks in the private caches (e.g., $8\times$ [25]).

Enlarging the size of the directory negatively impacts the directory access latency, area requirements, and energy consumption. Cuckoo directory [11] and SCD [23] report an acceptable provisioning size of $\sim 1.5\times$, yet it is still significantly sub-optimal. Given that area and power will be the most invaluable on-chip resources, it is important to seek for alternative, more scalable directory solutions.

## 2.2 Observations and Motivation

Instead of reducing directory evictions by enlarging the directory, this work aims at reducing directory-induced invalidations by just not enforcing them in certain (common) cases. The key observation is that the directory forces invalidations conservatively rather than selectively, oftentimes targeting blocks that are private to processors and by nature are not subject to coherence inconsistencies. In what follows, we describe the lifetime of cache blocks and directory entries, and we motivate our approach by showing that forcing the invalidation of private blocks is not only unnecessary, but more importantly, particularly detrimental.

The *life* of a block begins with a cache miss that loads new data into the private level of the cache hierarchy (Figure 2). The coherence protocol operates on the assumption that all data may become shared at any time; hence, an entry is always allocated at the directory to track the sharing status of the block and ensure coherence during its lifetime. The life of the block in the cache will end when a set conflict in the cache causes its replacement, or when a set conflict in the directory forces its invalidation.

Each block's life begins as private, where the block remains exclusive to the processor which initially loaded it. The life of the block may remain private during its lifetime (Figure 2-left), or may become shared (Figure 2-right). A block enters a "shared life" when a different processor references it, and remains shared till the end of its life.[1]

During the private life of a block, the corresponding directory entry experiences a period of "silence". This happens because all references to the block are issued by its exclusive processor and satisfied privately, thus never invoking the coherence protocol.[2]

Truly private blocks or those having "long" private life will therefore remain silent in the directory till their eviction from the directory. However, while the silent entries tracking private blocks will be moving relatively fast towards the LRU position of their directory sets and get victimized, the actual blocks in the cache might be highly active (hot). As a result, many evictions in the directory may force inval-

[1] We call a block "shared" regardless of how its temporal state changes. The shared status is lost when the block is removed from the directory.

[2] Assuming that the coherence protocol has an exclusive state (e.g., MESI).

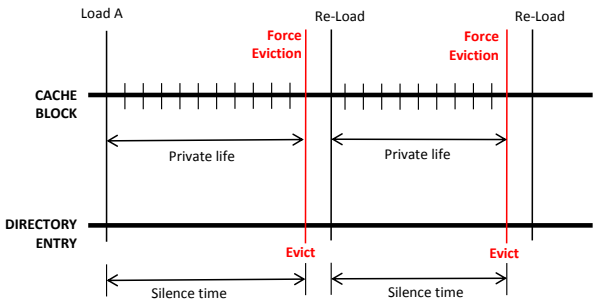



Figure 3: Forced invalidation of private blocks often results in misses that reload the same blocks.

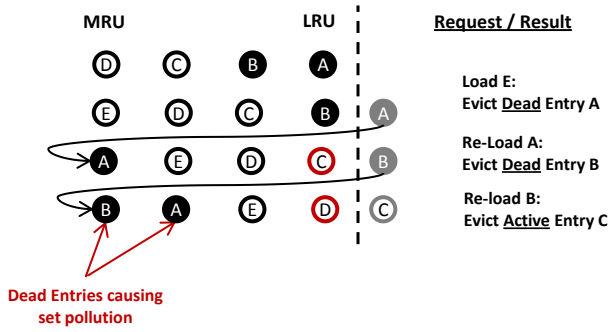



Figure 4: Pollution in directory sets due to reloads of prematurely evicted private blocks.

idations of private blocks that are still hot—hence likely to be recalled. Eventually, this could cause a sequence of long latency misses by the same processor reloading the same block, seriously hurting the overall system performance (Figure 3).

In addition to the significant impact they will have on the average miss latency, those misses will also disturb the locality that exists in the directory. In essence, reloading recently evicted private blocks creates an implicit promotional mechanism of silent (i.e., "dead") directory entries from the evicted position to the MRU (Figure 4). Maintaining dead entries in such a way is not only unnecessary, but will cause pollution in the directory sets, disturb the directory effectiveness and further hurt the overall performance.

The aforementioned observations present a highly undesirable case that could otherwise be avoided if private blocks were not invalidated by the directory. The potential opportunity is shown in Figure 5 which breaks down the directory-induced invalidation messages into those targeting private versus shared cached block, while indicating (with "x" mark) the percentage of blocks that will be reloaded soon after a directory-induced invalidation. The results, which are based on a $1/4\times$ under-provisioned directory size and LRU replacement policy, indicate that many directory evictions will invalidate private cached blocks

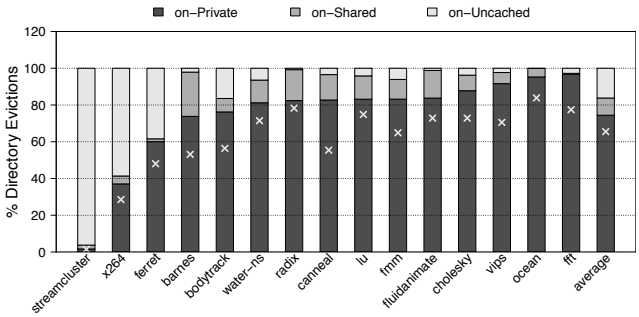



Figure 5: Breakdown of directory-induced invalidation targets.

(72%), while 80% of them will be requested again, causing extra cache misses and consequently degrade performance. As the size of the directory shrinks and as long as private data persist, the potential for improvement would be of major significance.

# 3 Stash Directory

Stash Directory evicts directory entries that track private blocks optimistically without forcing invalidations of corresponding cache blocks. Allowing the private cache blocks to continue enjoying their private life could significantly reduce cache misses and improve the system performance. Furthermore, the resulting permanent evictions of the directory's dead entries reduce the pollution in the directory and increase the utilization of the directory space.

## 3.1 Basic Organization and Operation

Stash Directory is a sparse directory design that allows conflicting entries tracking private blocks to be evicted without forcing invalidation of the actual block in the cache. In order to recognize whether a directory entry tracks a private or a shared block, each entry stores a bit indicating the lifetime sharing state of the tracked block. The bit is initially clear, indicating that the data block is private. The bit is set when the block changes from private to shared—an event that occurs when a second processor, other than the one that initially loaded the block, refers to it (i.e., issues a miss). The directory entry can recognize this event since it is always aware of the current sharers of the block. In fact, as the first processor has the exclusive ownership of the block, the directory can easily detect a private-to-shared event by just sensing a directory lookup from another processor. Eventually, when the directory entry reaches its eviction point, the bit is tested and, if it is clear (i.e., the cached block is still private), then no invalidation is enforced. Note that the extra bit is required since the existing sharer-vector in the directory entry is inadequate for identifying whether the block has been private to only one core during its lifetime. For example, it is possible for a nominally shared block to appear as private at different sharers at different points in time.

Stash Directory relinquishes cached blocks, leaving them "hidden" from the directory itself. Allowing such blocks to reside in the cache is safe as far as the blocks are guaranteed to be truly private to a specific processor. However, guaranteeing a truly private block is not always possible at run time. A block may transition from private to shared at any time; hence the "transitioning" miss may occur while the block is hidden from the directory, a situation we call a *false miss*. On a false miss, the coherence protocol sees a regular directory miss and falsely assumes that the referenced block is not present in any private cache. Therefore, the protocol forwards the miss to the shared LLC, expecting that it will be satisfied there.

Stash Directory delegates to the shared LLC and the coherence protocol the responsibility to detect a false miss and avoid inconsistency. The shared LLC keeps a bit for each cache block, indicating whether the block is possibly "cached" in the lower levels of the hierarchy. This bit is set by an explicit eviction notification message sent by the directory whenever it evicts entries without invalidating the cached copy. Therefore, a miss in the directory followed by a hit in the LLC having the "cached" bit set will immediately reveal a false miss. On detecting a false miss, the LLC controller will refuse to satisfy the miss and instead invoke back the coherence protocol to issue a broadcast request for discovering the most up-to-date copy of the block. After the miss is completed, the coherence protocol will register the block in the directory and reset the corresponding cached bit in the LLC.

As discussed, Stash Directory requires a shared LLC, which is typical in modern CMPs [9]. The additional storage required for the extra bits in the LLC is negligibly small ($<$0.2% of the cache size), is independent of the core count and scales perfectly with the cache size. Also note that an inclusion property is required between hidden blocks and the corresponding LLC entries. Fully inclusive LLCs (a design choice in most of Intel's CMP designs so far) inherently guarantee this property.[3] For non-inclusive caches [9], it is relatively easy to selectively enforce inclusiveness only for the blocks that are hidden. Specifically, (1) a cached block that is becoming hidden must be installed to the LLC if it is not already there (directory writes the victim's tag in LLC); and (2) a back-invalidation operation is selectively enforced only when the LLC block being evicted has the cached bit set. Lastly, handling a false miss requires standard support at the LLC controller to set/test the cached bit, and a standard broadcast read operation by the coherence protocol.

### 3.2 Handling LLC evictions

When an LLC block with its "cached" bit set is evicted, the cached (hidden) copy of that block must be removed from the private cache hierarchy. Given that LLC entries with their cached bit set have no directory information, the eviction's back-invalidation operation will have to be broadcast. This would significantly increase the frequency of broadcasting operations—and hence the bandwidth—since for every hidden private block, a broadcast operation is on the horizon. To avoid such situations, the cache system must implement clean eviction notifications (common in current commercial processors, e.g., AMD Opteron [9]). Accordingly, when a block is evicted from a private cache hierarchy, an eviction notification is sent to the directory to update the sharers to better reflect the sharing status of the block. If a notification does not find a corresponding entry in the directory, then it is implied that the block being evicted is a hidden private block and is currently tracked by the LLC. Therefore, the eviction notification will be forwarded to the LLC, and it will clear the cached status bit as there would be no longer a cached copy.

Clean eviction notifications are likely to clear the cached status of LLC blocks before they get evicted, hence reducing unnecessary broadcasts. In addition, we can further delay the cached LLC blocks from being evicted by allowing LLC sets to replace first those entries for which eviction notifications have been received. Further techniques to eliminate the negative effects of inclusiveness can be found in prior research [16].

### 3.3 Handling Shared Data

Stash Directory handles the evictions of entries tracking shared data blocks as a conventional sparse directory—it enforces invalidation on all cached copies. Although this may cause the invalidation of a block in more than one cores at once, it is preferred because, in contrast to private blocks, hiding shared blocks (especially migratory) could significantly increase the frequency of false misses as those blocks do not remain silent but are frequently referenced by different processors. In addition, since directory is sensitive to the temporal locality of blocks that are actively shared (recall Figure 2), victimized shared entries (LRU) are likely to track shared blocks which are either dead, or temporarily private, hence benefit from eviction. In the latter case, the eviction will give the the opportunity to a shared block to be reloaded as private (without a false miss) and benefit from re-entering the Stash Directory if it is to remain private. Although recent sparse directory implementations such as AMD HT Assist [9] suggest to explicitly avoid evicting shared entries, Stash Directory naturally avoids such evictions as it exploits the temporal behavior of actively shared blocks while reducing the pollution of directory sets from truly private entries.

### 3.4 Analysis on False Misses

**Probability of a false miss.** As described in Section 2, the transition of a block from private to shared may happen at

[3]The inclusion property forces a back-invalidation operation to all the sharers of the block being evicted.

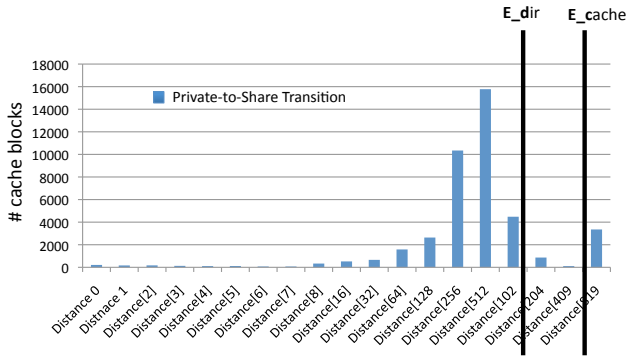



Figure 6: *The distribution of memory blocks based on their Private Lifetime (load-to-shared distance).* Distance is measured in memory references issued by the thread which loads the block. The distribution is largely an application attribute, but also depends on the relative progress of threads.

any time. A false miss occurs only when this transition takes place after the block is relinquished from the directory and before it is evicted from the cache, i.e., while the block is hidden. Theoretically, the longer the block's entry lives in the directory, the more likely that we can track the transition, and thus the lower probability of a false miss.

Figure 6 illustrates the distribution of memory blocks based on their private lifetime for a given application. Examining the distribution of blocks' private lifetime is useful because when compared with directory and cache eviction times, it can provide insights about the probability of reaching false misses. For example, if the point $E_c$ represents the average eviction time of a cache block, and $E_d$ the average eviction time of a directory entry, then the fraction of transitions appearing between the two points is the average number of memory references that would fall into a false miss. Hence, the average false miss probability is expressed as:

$$P(\textit{false miss}) = P_t(E_d < t < E_c) \tag{1}$$

where $P_t$ a probability distribution as the one depicted in Figure 6. We approximate $E_c$ and $E_d$ as follows.

Let *private-to-shared reuse distance* be the distance between two references on the same block, with the second being the one that transitions the block into shared. Then, we can derive the probability distribution of this distance as one appears in a cache ($p_t$), and as one appears in the directory ($q_t$). These distributions are basically the same as $P_t$ (Figure 6), except that $p_t$ may deviate (density shifted more toward the left), depending on whether and how the blocks are privately reused before they transition to shared. Then, assuming a fully-associative cache and directory [14],

$$P_t(t > E_d) = q_t(t > \alpha D) \tag{2}$$
$$P_t(t < E_c) = p_t(t < C) \tag{3}$$

where $D$ and $C$ is the directory and the cache size (in number of sets), respectively. $\alpha$ is a factor introduced to correct the mismatch between the distance as measured at the directory and the one that is seen at the processor's side. There are two reasons that create this mismatch: First, the directory absorbs memory references from all processors, hence a distance in the directory may appear larger relative to what a processor experiences. Second, only memory references that will invoke the coherence protocol or miss in the processor's cache will reach the directory, and therefore many will be filtered out. If we assume that all processors reference memory blocks at the same rate and that $P_s$ is the probability of a reference reaching the directory, then a reasonable approximation for $\alpha$ would be:

$$\alpha \approx \frac{1}{\text{core count}} \times \frac{1}{P_s} \tag{4}$$

The final expression for the false miss probability is

$$P(\textit{false miss}) = q_t(t > \alpha D) + p_t(t < C) - 1 \tag{5}$$

which is derived from Equations (1), (2), and (3), and is dictated by the application- and machine- specific properties of $q_t$, $p_t$, and $P_s$, the cache and directory parameters, and the core count.[4]

**Upper bound for false misses.** The worst case with regard to false misses occurs if all private-to-shared references miss in the directory and hit in a cache. In such case, the total actual number of false misses will be equal to the total number of private-to-shared references. The transition from private to shared can occur at least once for each shared block, but it may also occur more times as the same block may be evicted and reloaded again. A reloaded block always appears as private and thus creates a possibility for a new private-to-shared transition. Thus, the upper bound can be expressed as

$$\#\textit{ false misses} < \#\textit{ shared blocks} \times S \tag{6}$$

where $S$ is the average number of times a shared block will go through a private-to-shared transition, which we could approximate by calculating the probability of a shared block falling into a reuse distance larger than $E_c$ and $E_d$.

For shared data sets that are relatively small and frequently accessed (i.e., have short reuse distance), the $S$ factor could be very small and the bound closer to the actual number of application's shared blocks. In such cases, the worst case for false misses is largely an application-specific attribute that can be easily evaluated before any architectural exploration. A comparison between this worst-case bound and experimental measurement of false misses is presented later in Section 5.1.

[4]For set-associative caches, the expression can be approximated based on the same probability distributions [4, 14].

| *Model* | *Configuration Values* |
|---|---|
| Cores | 16 in-order cores, tiled, 2-issue width. |
| L1 Cache | 32 KB D + 32KB I, 4-way, 1-cycle tag latency |
| L2 Cache | 256 KB private, 8-way , 3-cycle tag latency |
| L3 Cache | 16MB shared (NUCA), 16-way, 5/30-cycle tag/data |
| Coherence | MESI protocol, clean eviction notification. |
| Directory | 2x-1/8x provisioned, 8-way, LRU. |
| NoC | 4×4 2D mesh, 2-stage routers, wormhole-switched, determ. X-Y routing, Ack/Nack flow control. |
| Main mem. | 70-cycle latency, 4KB pages, 64-entry 4-way TLB |

Table 1: Machine Architecture Configuration.

| BENCHMARK | INPUT | BENCHMARK | INPUT |
|---|---|---|---|
| *barnes* | 16K (particles) | *bodytrack* | simsmall |
| *cholesky* | tk25.O | *canneal-* | simsmall |
| *fft* | 256K (points) | *ferret* | simsmall |
| *fmm* | 16K (particles) | *fluidanimate* | simsmall |
| *lu* | 512 (matrix) | *streamcluster* | simsmall |
| *ocean* | 258 (grid) | *vips* | simsmall |
| *radix* | 8M (keys) | *x264* | simsmall |
| *water-ns* | 512 (mol.) | | |

Table 2: Workloads and Input Data Sets.

In summary, the analysis suggests that the rate of false misses can be approximated before performing time-consuming evaluations, and can provide valuable hints towards workload characterization and design exploration.

# 4 Evaluation Methodology

## 4.1 System Model

We simulate a 16-core tiled CMP system using x86-64 memory traces collected with PIN [19]. The traces are fed into a detailed cycle-accurate cache and interconnect model. Each tile has an in-order single-threaded core with private L1/L2 cache, a shared L3 bank, and a directory slice. The tiles are interconnected through a 4×4 2D mesh network-on-chip (NoC). All private caches are kept coherent by a MESI coherence protocol and a distributed, address-interleaved directory. Table 1 summarizes the main architecture configuration parameters.

## 4.2 Workloads

We study multithreaded workloads from the splash2 and parsec suites [5, 27] with input sizes as listed in Table 2. Detailed simulations are performed during both parallel and sequential phases of the workloads and statistics are collected at the end of a parallel phase. We use all available processor cores by spawning 16 concurrent threads in all experiments. For stable and repeatable measurements, we prevent thread migration by binding each thread to the first touched core. For workloads that create more threads than CPUs we enforce deterministic scheduling.

## 4.3 Evaluated Schemes

We evaluate Stash Directory (or "*Stash*" for short) relative to a conventional sparse directory implementation [12] ("*Sparse*" for short). We also compare against the state-of-the-art approach to reducing the number of required directory entries, which deactivates coherence for private blocks [10]. We refer to the latter scheme as *PDC* (standing for Private Deactivation Coherence). More schemes are qualitatively compared in Section 5. All the evaluated designs share the same basic structure—a set-associative array with each entry holding a cache tag and a sharer vector. The number of tags (entries) in the directory defines the maximum number of addresses that can be represented in the directory. This number, essentially the storage capacity of the directory, is reported in relation to the total lines in the tracked caches; i.e., 1× when it is equally-provisioned, 1.5×, 2×, etc. when over-provisioned, and 1/2× 1/4× etc. when under-provisioned.

All designs use a full sharer vector per entry to record the sharing information. Note that the sharer set can be encoded in various ways, and there is rich literature on tackling the scalability limitations of a full-map sharer vector [1, 6–8, 12]. It is important to remember that the sharer information encoding is independent of how the evaluated schemes are organized. Directory access latency and power are modeled after the same technology assumed for the private L2 caches.

Stash Directory is augmented with an extra bit per entry denoting whether the block being tracked is private or shared, and the LLC with an extra bit per entry ("cached" bit). References or replacements in the LLC will trigger a broadcast operation if the cached bit is set. Eviction notifications that reach the LLC will clear the cached bit.

For the PDC scheme we employ mechanisms that classify memory pages into private or shared, and we deactivate the coherence protocol for those memory requests sorted as private. The classification and detection mechanism requires modifications to the TLB miss handler and the page table, which we implement as described in [10].

# 5 Evaluation Results

## 5.1 Impact of Directory Size on Cache Performance

By eliminating many directory-induced invalidations, Stash prevents the increase in cache misses and improves cache performance. Figure 7 shows the increase in cache miss rate when employing a conventional sparse directory in contrast to Stash. The first four bars in each workload correspond to Sparse, starting with a 2× provisioning size till down to $1/4\times$. The last bar shows the result of Stash, when it is under-provisioned to $1/4\times$. As the results illustrate, in almost all cases Sparse severely hurts the cache performance if not over-provisioned. In contrast, Stash is able to significantly alleviate that impact, or in many cases completely elide it. For almost all applications, a $1/4\times$ provisioned Stash can perform as well as a 2× provisioned Sparse. Consequently, Stash can reduce the space requirements by eight

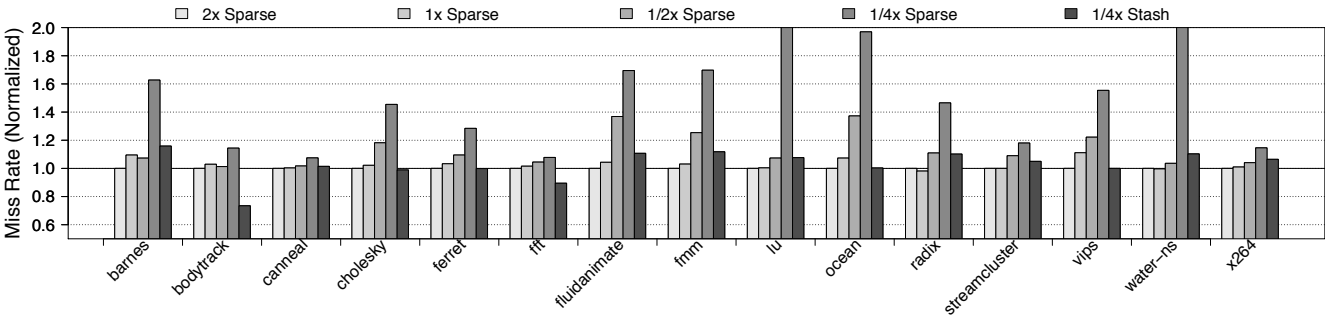



Figure 7: *Cache Miss Rates for different Sparse and Stash Directories.* Results are normalized to a $2\times$ Sparse Directory. Cache misses count the accesses that miss in both L1 and L2.

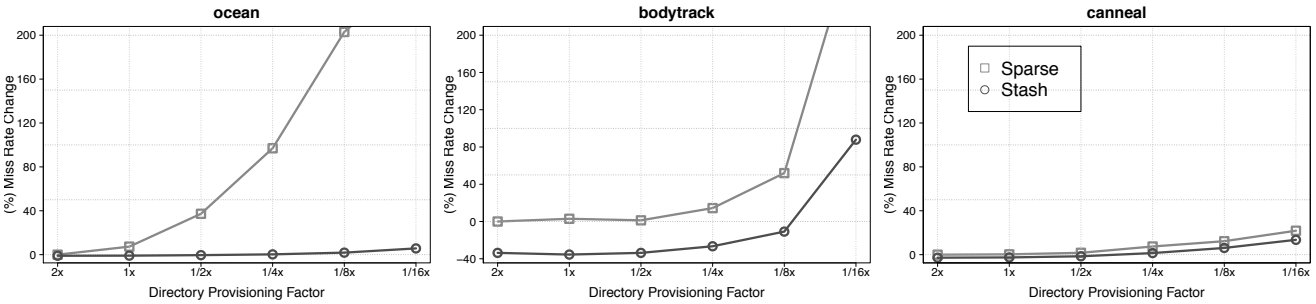



Figure 8: *Examples of the Cache Miss Rate Sensitivity to the Directory Provisioning Size.*

times (i.e., to $1/8$ of the original size) with essentially no impact on cache performance. In contrast, a Sparse directory with such a small size would have increased the miss rate significantly (e.g., by more than 40% for radix). In some cases Stash can perform better than $2\times$ Sparse (e.g., bodytrack, fft), indicating that even with $2\times$ size, a Sparse can have a negative impact on performance.

Following how the miss rate changes as the capacity of the directory varies can offer further insight on how the Stash scheme performs. In the case of the conventional Sparse, decreasing the directory will always create more conflicts, and since every conflict enforces invalidation, the miss rate will always suffer more. In the case of Stash, however, the cache performance won't be affected unless the directory starts evicting entries that track shared blocks. This situation will possibly occur when the directory size becomes smaller than that of the working *shared* data set size that is usually much smaller than that of the overall working set. The first example in Figure 8 (ocean) illustrates a case in which cache performance remains unaffected even with heavily under-provisioned Stash capacity. In this example, Stash can be as small as $1/16\times$ without impacting cache performance. We found that such a significant under-provisioning opportunity exist in 5 out of the 15 studied workloads (fft, ocean, radix, vips, x264). By decreasing further the directory capacity, the miss rate will eventually become sensitive—a point which could be associated with the shared data's working set size of the application. Typically, most applications will naturally demonstrate some level of sensitivity as shared entries in directory will conflict with newly loaded shared or private entries. Stash alleviates the impact compared to Sparse as illustrated by the fluidanimate's example in Figure 8. In this case, Stash Directory degrades the cache performance in a slower rate compared to Sparse. Note that in this example a Stash with $1/8\times$ capacity is able to keep the miss rate lower than the large, $2\times$ Sparse directory. Lastly, applications with poor shared data locality or stream-like references to private data may prevent Stash from being effective. Applicatons with relatively hight miss rate, such as canneal and streamcluster are not much sensitive to the changing directory capacity (e.g., Figure 8).

**False misses.** False misses trigger broadcast operations, which are more demanding in terms of bandwidth. If false misses occur frequently, then they could threaten the overall energy and performance of a system. Our analysis in Section 3.4 shows that the probability of a false miss and an upper bound can be easily approximated and could be fairly small. Figure 9 shows experimental results for the amount of false misses, for all the applications in our study and for different directory sizes. Also, for each application, we draw the upper bound given by Equation (6) as-

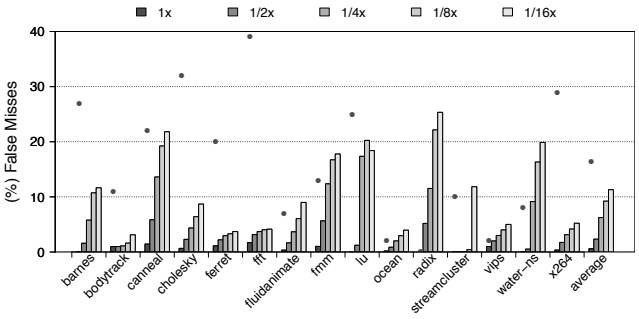



Figure 9: *False misses* relative to the total misses for different stash directory sizes. The dots show a theoretic upper bound approximation.

suming $S = 1.01$ (shown with a dot above the bar—only for the $1/4\times$ case). The results show that the number of false misses is around 6% on average for the $1/4\times$ case, and they range from 0% to 25% in the worst case. As expected, false misses are relatively sensitive to the directory size. As we will show in the rest of this section, these false misses do not raise significant concerns and do not adverse the improvements that Stash offers in overall.

## 5.2 Performance Comparisons

In Figure 10 we compare bandwidth demands, execution time and energy consumption of $1/4\times$ Sparse, PDC and Stash directories. All the results are normalized to a $2\times$ base sparse directory.

**Bandwidth requirements.** Figure 10a shows the NoC traffic generated with each scheme, measured as the total number of bytes transmitted during execution. In Stash, we break down the additional traffic generated from broadcasts into broadcast traffic due to false misses, and due to LLC evictions of hidden blocks (marginal difference). Despite the added broadcast traffic, Stash has bandwidth requirements similar to the $2\times$ Sparse, while when compared to equal-size Sparse (1/4× Sparse), it reduces the overall NoC traffic by 50% on average. The improvements come mainly from the reduction in requests for data (i.e., misses), as well as the reduction in directory-induced invalidation and their associated writebacks.

Compared to PDC, Stash achieves slighly less bandwidth savings on average; This is due to the large amount of coherence msgs saved when the data classification mechanism is in a good effect for PDC. In cases such as water-ns, however, Stash can be significantly more bandwidth-efficient, achieving improvements of up to 35% over PDC. The results show also that broadcasting in Stash adds a 0% to 10% of extra traffic. Broadcasting can be completely eliminated if we replace the cached bit at LLC with the actual ID of the core holding the block. This will cause however more redundant storage in LLC and therefore is a design choice that should be further evaluated.

**Overall performance.** Figure 10b shows execution time achieved by the studied schemes. On average, 1/4× Stash can improve performance by 16% compared to Sparse, and by 2% compared to PDC. In contrast to Sparse, Stash tolerates well the under-provisioned directory size (1/4×) and performs equal to the $2\times$ Sparse directory. This suggests that Stash can reduce space requirements to as low as $1/8$ that of a conventional directory without compromising performance. Considering the strong correlation of performance to the cache miss rate, we expect that Stash can often sustain performance even for smaller directory sizes (e.g., ocean in Figure 8),

The quantitative comparison between PDC and Stash does not lead to a clear winner on average; however it highlights the importance of Stash versus PDC in design decision making. Stash is a fundamentally different approach as it addresses the problem of handling private data in an *aggressive* rather than *conservative* manner; Stash Directory evicts silently all entries that appear to be *optimistically* private, eliminating *all* directory-induced invalidations on private blocks, and relying in a simple mechanism to discover false cases. In contrast, PDC explicitly aims at saving space by preventing private blocks from entering the directory; however, it can do so only for *some fraction* of them as its detection mechanism is conservative and cannot tolerate incorrect speculations. Essentially, Stash is expected to be highly effective in applications where memory pages have mixed private and shared data, as well as good temporal locality, because it will gradually remove all private blocks from the directory. However, in memory-bound applications (e.g., canneal), Stash performs less effectively, reaching equal to or worse than PDC performance levels.

Furthermore, Stash is a simpler design approach in that it is implemented at the last level of the cache hierarchy, and is decoupled from in-core structures such as TLBs—which are already quite complex and heavily accessed. Stash is in turn a transparent and effective optimization that works independently of the type of cores that are plugged in, and of the type of system software that is running. In a further perspective, Stash and PDC are orthogonal techniques, and therefore can be combined to provide the best of both or even additional improvements.

**Energy efficiency**. Reduction of bandwidth demands and execution time directly translates to energy savings. To assess the energy consumption, we use an intuitive model that considers the dynamic energy consumed in the directory, the interconnect, and the L2/L3 cache lookups, including coherence requests. The energy consumed per directory and cache access is estimated using CACTI [15] assuming a 32nm technology. For interconnect, we assume that the energy consumption is proportional to the amount of data transferred and that lookups on network routers consume four times more power than link traversals. . Note that we

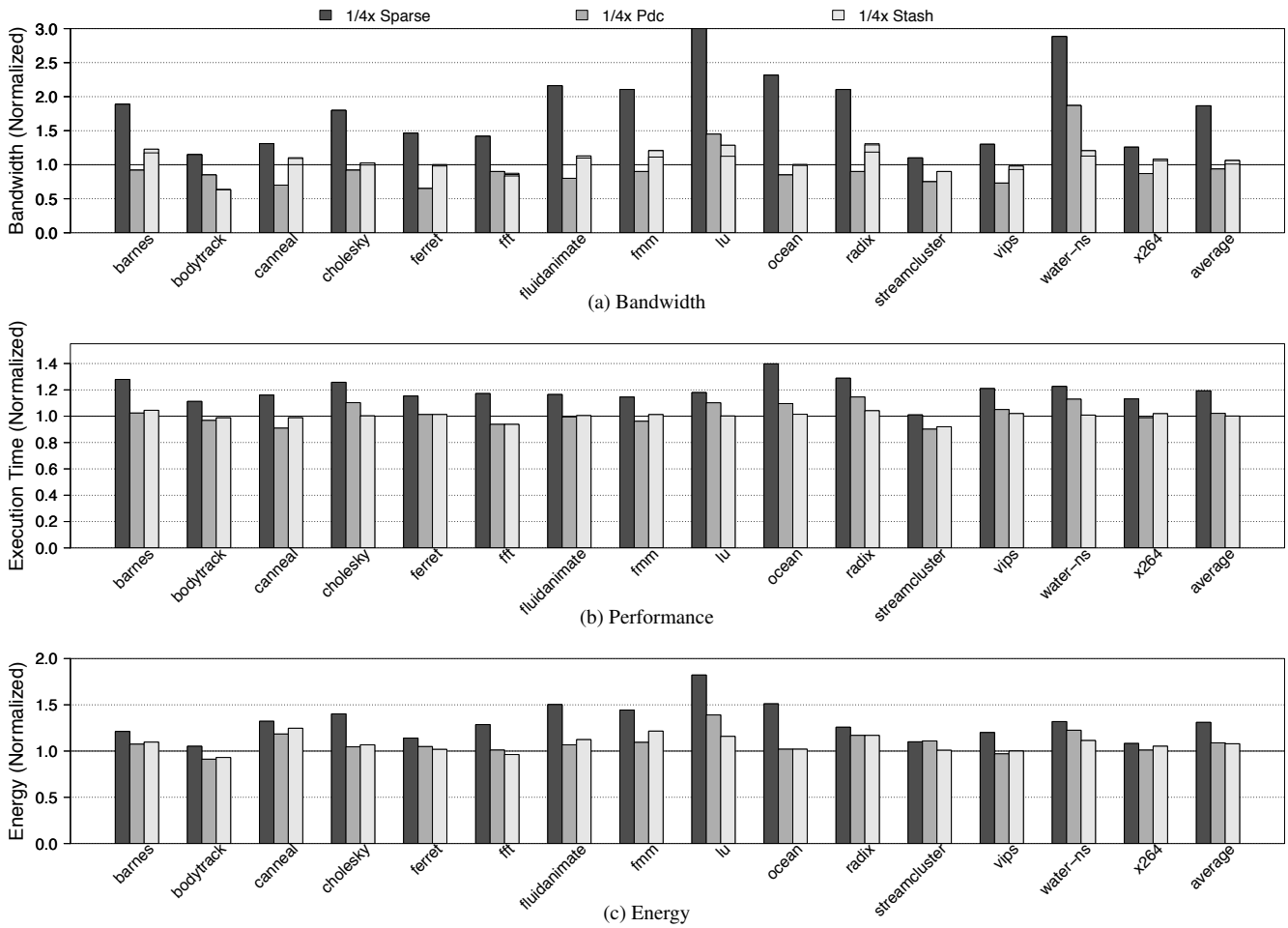



Figure 10: Comparison of $1/4\times$ under-provisioned directory organizations. Results are normalized to $2\times$ Sparse.

do not account static energy or energy consumed in other components. Figure 10c presents the results for the different schemes. Stash Directory achieves partial reductions in energy dissipation in the cache hierarchy by 18% compared to Sparse, and compares equally to the PDC approach.

### 5.3 Scalability

A feasible directory design for coherent many-core CMPs must scale well across an increasing number of cores. Area and energy requirements must be kept minimum, as they will present scarce resources in many-core chips, while performance should be sustainably unaffected by the existence of the directory. Stash inherits the power efficiency of sparse directories, as well as the numerous scalable sharer encoding techniques. Therefore, what remains essential to examine is whether the benefits of the Stash design extend as the core count increases, continuing to offer storage improvements without affecting the performance or energy of the system.

Figure 11 presents performance and bandwidth scaling trends for the three different organizations. To scale out the evaluation to many cores, we spawn up to as many threads as possible on a native machine, trace the memory references of each thread and the instructions between them, and feed the trace to a cache simulator. The results show that Stash Directory scales well on average. Cache performance gets better as the core count increases mainly because of the relation of Stash with the size and locality of shared data sets—which are basically independent of the core count. Despite the broadcasting operations (that are inherently non-scalable), Stash bandwidth remains relatively insensitive, suggesting that the amount of broadcasts is not significant enough to impact bandwidth and energy negatively. PDC follows Stash closely, but for high core count its overall performance is less robust. This is due to a degraded effectiveness in classifying the private blocks. In the case of the base directory, the core growth will directly increase the directory-induced invalidations, which will in turn degrade the performance severely.

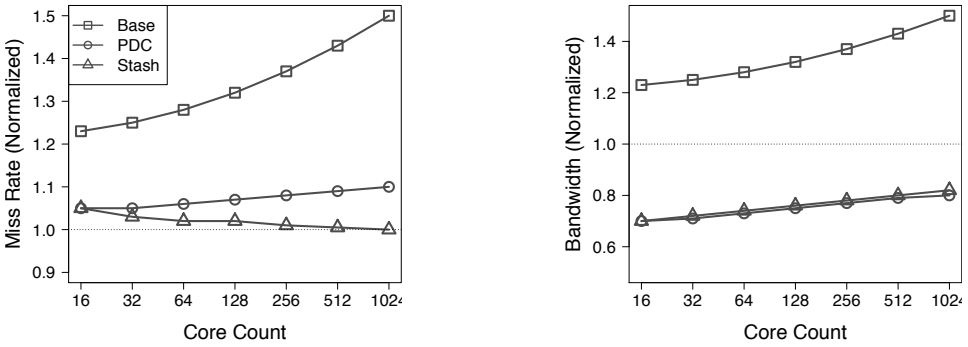



Figure 11: *Many-core Scalability:* Proxies for performance (left) and energy (right). The measurements assume that the total cache size increases in proportion to the core count and the directory size remains $1/4\times$ the cache size.

## 6 Related Work

There is a large body of work addressing the scalability challenges of cache coherence directory. This section discusses the most related prior work, focusing on studies looking into the storage inefficiencies in the set-associative directory structure.

**Duplicate-tag and tag-less organizations.** Duplicate-tag directories, used in Piranha [3] and Niagara 2 [21], can keep one-to-one correspondence to cache blocks, and do not require keeping extra sharing information. Their storage requirements are equal to the aggregated L1 tags. However, their associativity must be equal to the cache associativity $\times$ core count, which adds significant overheads per lookup and makes this organization far from scalable. "Tagless directory" [28] uses instead a set of Bloom filters to test for tag membership and does not rely on fully duplicated tags. This makes it even more storage efficient than duplicate-tags (especially for small core counts) and also improves energy dissipation; however, similar to the duplicate-tag directory, it has a quadratic growth in chip energy dissipation for the aggregate of all tag-less directory slices and fails to scale beyond 128 cores.

**Sparse directories.** Sparse directories [12] are more energy efficient than the duplicate-tag directory since they have a relatively low associativity. However, their storage must be over-provisioned to reduce conflicts in directory entries as they may negatively affect performance. Hence, their design usually incurs significant area overheads. Apart from that, Sparse directories must keep full sharer vectors for each cached block, which adds storage overheads that do not scale with core count. This limitation has been addressed by numerous prior works [1, 6–8, 12, 17, 18, 26, 28, 29] and is not in the scope of this work.

To reduce the over-provisioning requirements, recent proposals such as Cuckoo [11] and SCD [23] directories resolve set conflicts using multi-hashing indexing and insertion. Although these techniques increase the space efficiency, they do not take into account any characteristics unique to the coherence directory behavior and hence are limited in their potential. Furthermore, the aforementioned techniques require progressive replacement/insertion operations, and thus they affect the average latency of the directory lookup. The relaxed inclusion property of Stash Directory is orthogonal to the set conflict resolving techniques used by Cuckoo or SCD, and thus can be combined.

An alternative approach for increasing the effectiveness of the directory space is to deactivate the coherence for private memory blocks [10]. We compare this approach against Stash in the previous sections, referred as PDC. In summary, PDC depends on a page-granularity data classification mechanism [13] that limits its potential. Also, it relies on page table modifications, which could be hard to justify for relatively minor microarchitecture performance improvements. Stash is instead a fairly simple and transparent scheme that can be easily adopted in certain designs.

Lastly, Alisafaee [2] recently proposed a scheme called "spatiotemporal coherence tracking" (SCT) that saves directory space by tracking temporarily private data in a coarse-grain fashion. Temporarily private data are shared data that appear private for long periods, i.e., data accessed and cached by a single core most of the time. The author evaluates SCT against workloads that are dominated by shared data footprints and shows that SCT can greatly benefit from the proposed compaction technique. Specifically, SCT reports $1/2\times$ effective provisioning.

In comparison, Stash does not directly optimize for shared data and thus applications with dominant shared sets may not benefit as much as with SCT. However, Stash can exploit effectively the (even limited) private data of such applications, while PDC is less likely to be effective is such a case. Also, Stash will sometimes treat temporarily private blocks as truly private and benefit from them as SCT would. For example, a block that has a long private life before transitioning to share will be silently removed from the directory and remain hidden till the time the transition. While SCT compacts entries in order to save space and requires fairly complicated mechanisms and policies to manage the compacted regions, Stash simply removes entries

out of the directory without further storage considerations. For workloads with more private data, Stash is expected to be substantially more storage- and cost-efficient than SCT. The two approaches may be combined synergistically to offer a robust solution across many application domains.

## 7 Conclusions

This work proposed and evaluated *Stash Directory*, a novel sparse directory design that requires significantly smaller storage than conventional directories. Stash Directory allows directory entries tracking private blocks to be evicted without invalidating the corresponding cached blocks. As a result, it eliminates performance loss due to premature data block invalidations in private caches, and at the same time reduces the pollution in the directory sets that are caused by subsequent recalls of the same blocks. When private memory blocks are dominant in caches, which is often the case with parallel and multiprogramming workloads, Stash Directory offers an effective, scalable and transparent solution that has nearly constant power and area utilization regardless of the core count.

## References


[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of the Annual Int'l Symp. on Computer Architecture*, ISCA, 1988.

[2] M. Alisafaee. Spatiotemporal coherence tracking. In *Proc. of the Annual Int'l Symp on Microarchitecture*, 2012.

[3] 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 *Proc. of the Annual Int'l Symp. on Computer Architecture*, ISCA, 2000.

[4] K. Beyls and E. H. D. Hollander. Reuse distance as a metric for cache behavior. In *Proc. of the IASTED Conf. on Parallel and Distributed Computing and Systems*, 2001.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. PACT, 2008.

[6] D. Chaiken, J. Kubiatowicz, and A. Agarwal. Limitless directories: A scalable cache coherence scheme. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 1991.

[7] G. Chen. Slid - a cost-effektive and scalable limited-directory scheme for cache coherence. In *Proc. of the Int'l PARLE Conf. on Parallel Architectures and Languages Europe*, PARLE, 1993.

[8] J. H. Choi and K. H. Park. Segment directory enhancing the limited directory cache coherence schemes. In *Proc. of the Int'l Symp. on Parallel Processing and Parallel and Distributed Processing*, IPPS/SPDP, 1999.

[9] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 2010.

[10] 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 *Proc. of the Annual Int'l Symp. on Computer Architecture*, 2011.

[11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *Proc. of the Int'l Symp. on High Performance Computer Architecture*, HPCA, 2011.

[12] A. Gupta, W. dietrich Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *In Int'l Conf. on Parallel Processing*, 1990.

[13] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *Proc. of the Annual Int'l Symp. on Computer architecture*, ISCA, 2009.

[14] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 1989.

[15] http://quid.hpl.hp.com:9081/cacti/. CACTI 5.3.

[16] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. MICRO, 2010.

[17] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel. Waypoint: scaling coherence to thousand-core architectures. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, PACT, 2010.

[18] C. Li, H. Wang, Y. Xue, X. Zhang, and D. Wang. Fast hierarchical cache directory: A scalable cache organization for large-scale cmp. In *Proc. of the Int'l Conf. on Networking, Architecture, and Storage*, NAS, 2010.

[19] 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 *Proc. of the Conf. on Programming Language Design and Implementation*, PLDI, 2005.

[20] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 2012.

[21] S. Microsystems. Ultrasparc t2 supplement to the ultrasparc architecture. Technical report, 2007.

[22] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proc. of the Annual Int'l Symp. on Computer Architecture*, ISCA, 1990.

[23] D. Sanchez and C. Kozyrakis. Scd: A scalable coherence directory with flexible sharer set encoding. In *Proc. of the Int'l Symp. on High-Performance Computer Architecture*, 2012.

[24] R. Simoni. Cache coherence directories for scalable multiprocessors. Technical report, 1992.

[25] R. Singhal. Inside intel next-generation nehalem microarchitecture. In *Hot Chips 20*, Stanford, CA,, 2008.

[26] D. A. Wallach. Phd: A hierarchical cache coherent protocol, 1992.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. ISCA, 1995.

[28] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proc. of the Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2009.

[29] H. Zhao, A. Shriraman, and S. Dwarkadas. Space: sharing pattern-based directory coherence for multicore scalability. PACT, 2010.