SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding

Daniel Sanchez and Christos Kozyrakis
Stanford University
{sanchezd, kozyraki}@stanford.edu

Abstract

Large-scale CMPs with hundreds of cores require a directory-based protocol to maintain cache coherence. However, previously proposed coherence directories are hard to scale beyond tens of cores, requiring either excessive area or energy, complex hierarchical protocols, or inexact representations of sharer sets that increase coherence traffic and degrade performance.

We present SCD, a scalable coherence directory that relies on efficient highly-associative caches (such as zcaches) to implement a single-level directory that scales to thousands of cores, tracks sharer sets exactly, and incurs negligible directory-induced invalidations. SCD scales because, unlike conventional directories, it uses a variable number of directory tags to represent sharer sets: lines with one or few sharers use a single tag, while widely shared lines use additional tags, so tags remain small as the system scales up. We show that, thanks to the efficient highly-associative array it relies on, SCD can be fully characterized using analytical models, and can be sized to guarantee a negligible number of evictions independently of the workload.

We evaluate SCD using simulations of a 1024-core CMP. For the same level of coverage, we find that SCD is 13× more area-efficient than full-map sparse directories, and 2× more area-efficient and faster than hierarchical directories, while requiring a simpler protocol. Furthermore, we show that SCD's analytical models are accurate in practice.

1. Introduction

As Moore's Law enables chip-multiprocessors (CMPs) with hundreds of cores [15, 28, 30], implementing coherent cache hierarchies becomes increasingly difficult. *Snooping* cache coherence protocols work well in small-scale systems, but do not scale beyond a handful of cores due to their large bandwidth overheads, even with optimizations like snoop filters [18]. Large-scale CMPs require a *directory-based* protocol, which introduces a *coherence directory* between the private and shared cache levels to track and control which caches share a line and serve as an ordering point for concurrent requests. However, while directory-based protocols scale to hundreds of cores and beyond, implementing directories that can track hundreds of sharers efficiently has been problematic. Prior work on thousand-core CMPs shows that hardware cache coherence is important at that scale [18, 20], and hundred-core directory-coherent CMPs are already on the market [30], stressing the need for scalable directories.

Ideally, a directory should satisfy three basic requirements. First, it should maintain sharer information while imposing small area, energy and latency overheads that scale well with the number of cores. Second, it should represent sharer information accurately — it is possible to improve directory efficiency by allowing inexact sharer information, but this causes additional traffic and complicates the coherence protocol. Third, it should introduce a negligible amount of directory-induced invalidations (those due to limited directory capacity or associativity), as they can significantly degrade performance.

Proposed directory organizations make different tradeoffs in meeting these properties, but no scheme satisfies all of them. Traditional schemes scale poorly with core count: Duplicate-tag directories [2, 29] maintain a copy of all tags in the tracked caches. They incur reasonable area overheads and do not produce directory-induced invalidations, but their highly-associative lookups make them very energy-inefficient with a large number of cores. Sparse directories [13] are associative, address-indexed arrays, where each entry encodes the set of sharers, typically using a bit-vector. However, sharer bit-vectors grow linearly with the number of cores, making them area-inefficient in large systems, and their limited size and associativity can produce significant directory-induced invalidations. For this reason, set-associative directories tend to be significantly oversized [10]. There are two main alternatives to improve sparse directory scalability. Hierarchical directories [31, 33] implement multiple levels of sparse directories, with each level tracking the lower-level sharers. This way, area and energy grow logarithmically with the number of cores. However, hierarchical organizations impose additional lookups on the critical path, hurting latency, and more importantly, require a more complex hierarchical coherence protocol [31]. Alternatively, many techniques have been explored to represent sharer sets inexactly through coarse-grain bit-vectors [13], limited pointers [1, 6], Tagless directories [35] and SPACE [36]. Unfortunately, these methods introduce additional traffic in the form of spurious invalidations, and often increase coherence protocol complexity [35].

In this paper, we present the Scalable Coherence Directory (SCD), a novel directory scheme that scales to thousands of cores efficiently, while incurring negligible invalidations and keeping an exact sharer representation. We leverage recent prior work on *efficient highly-associative caches* (ZCache [25] and Cuckoo Directory [10]), which, due to their multiple hash functions and replacement process, work in practice as if replacement candidates were selected randomly, independently of the addresses tracked [25]. We exploit this property to design and analyze SCD:

- First, we recognize that, to be scalable, a directory implementation only needs the number of bits *per tracked sharer* to scale gracefully (e.g., remaining constant or increasing logarithmically) with the number of cores. SCD exploits this insight by using a *variable-size sharer set representation*: lines with one or few sharers use a single directory tag, while widely shared lines use additional tags. We propose a hybrid pointer/bit-vector organization that scales logarithmically and can track tens of thousands of cores efficiently. While conventional set-associative arrays have difficulties with this approach, highly-associative zcache arrays allow SCD to work.
- We develop analytical models that characterize SCD and show how to size it. First, we show that for a given occupancy (fraction of directory capacity used), SCD incurs the same number of directory-induced invalidations and average number of lookups, independently of the workload. Second, different workloads impose varying capacity requirements, but the worst-case capacity requirement is bounded and small. Hence, directories can be built to guarantee a negligible number of invalidations and a small average number of lookups in all cases, guaranteeing performance and energy efficiency with just a small amount of overprovisioning (around 5-10% depending on the requirements, much smaller than what is required with set-associative arrays [10]). These results are useful for two reasons. First, they enable designers to quickly size directories without relying on empirical results and extensive simulations. Second, they *provide guarantees on the interference introduced by the shared directory*, which is paramount to achieve performance isolation among multiple competing applications sharing the chip (CMPs need a fully shared directory even if they have private last-level caches). This analytical characterization also applies to sparse directories implemented with highly-associative caches (Cuckoo Directories), which prior work has studied empirically [10].

We evaluate SCD by simulating CMPs with 1024 cores and a 3-level cache hierarchy. We show that, for the same level of provisioning, SCD is 13× more area-efficient than sparse directories and 2× more area-efficient than hierarchical organizations. SCD can track 128MB of private cache space with a 20MB directory, taking only 3% of total die area. Moreover, we show that the analytical models on invalidations and energy are accurate in practice, enabling designers to guarantee bounds on performance, performance isolation, and energy efficiency.

2. Background and Related Work

In this section, we provide the necessary background for SCD, reviewing previously proposed directory organizations and efficient highly-associative arrays, which SCD relies on.

2.1. Directory Organizations

Cache coherence is needed to maintain the illusion of a single shared memory on a system with multiple private caches. A coherence protocol arbitrates communication between the private caches and the next level in the memory hierarchy, typically a shared cache (e.g., in a CMP with per-core L2s and a shared last-level cache) or main memory (e.g., in multi-socket systems with per-die private last-level caches). In this work we focus on *directory-based*, *write-invalidate* protocols, as alternative protocols scale poorly beyond a few private caches [14]. These protocols use a *coherence directory* to track which caches share a line, enforce write serialization by invalidating or downgrading access permissions for sharers, and act as an ordering point for concurrent requests to the same address. Implementing a directory structure that scales to hundreds of sharers efficiently has been problematic so far. We now review different directory organizations, with a focus on comparing their scalability. Table 1 summarizes their characteristics.

Traditional directory schemes do not scale well with core count. *Duplicate-tag* directories maintain a full copy of all the tags tracked in the lower level. Their area requirements scale well with core count, but they have huge associativity requirements (e.g., tracking 1024 16-way caches would require 16384 ways), so they are limited to small-scale systems [2, 29]. In contrast, *sparse directories* [13] are organized as an associative array indexed by line address, and each directory tag encodes the set of sharers of a specific address. Sparse directories are energy-efficient. However, due to their limited associativity, sparse directories are often forced to evict entries, causing *directory-induced invalidations* in the lower levels of the hierarchy. This can pose large performance overheads and avoiding it requires directories that are significantly overprovisioned [10].

The encoding method for the sharer set is a fundamental design choice in sparse directories. *Full-map* sparse directories encode the sharer set exactly in a bit-vector [13]. They support all sharing patterns, but require storage proportional to the number of cores, and scale poorly beyond a few tens of cores. Alternatively, sparse directories can use a compressed but inexact encoding of the sharer set. Traditional alternatives include *coarse-grain sharer bit-vectors* [13], and *limited pointer* schemes, in which each entry can hold a small number of sharer pointers, and lines requiring more sharers either cause one of the existing sharers to be invalidated, a broadcast on future invalidations and downgrades [1, 20], or trigger an interrupt and are handled by software [6]. Inexact sharer set schemes trade off space efficiency for additional coherence traffic and protocol complexity.

In contrast to these techniques, *hierarchical sparse directories* [12, 31, 33] allow an exact and area-efficient representation of sharer sets. Hierarchical directories are organized in multiple levels: each first-level directory encodes the sharers of a subset of caches, and each successive level tracks directories of the previous level. A two-level organization can scale to thousands of cores efficiently. For example, using 32 first-level directories and one (possibly banked) second-level directory, we can track 1024 caches using 32-bit sharer vectors, or 4096 caches using 64-bit vectors. However, hierarchical directories have two major drawbacks. First, they require several lookups on the critical path, increasing directory latency and hurting performance. Second, multi-level

Scheme	Scalable area	Scalable energy	Exact sharers	Dir-induced invalidations	Extra protocol complexity	Extra latency
Duplicate-tag	Yes	No	Yes	No	No	No
Sparse full-map	No	Yes	Yes	Yes	No	No
Coarse-grain/limptr	No	Yes	No	Yes	Small	No
Hierarchical sparse	Yes	Yes	Yes	Yes	High	Yes
Tagless	No	No	No	Yes	Medium	No
SPACE	No	Yes	No	Yes	Small	No
Cuckoo Directory	No	Yes	Yes	Negligible	No	No
SCD	Yes	Yes	Yes	Negligible	No	No

Table 1: Qualitative comparison of directory schemes. The first three properties are desirable, while the last three are undesirable.

coherence protocols are more complex than single-level protocols, and significantly harder to verify [7, 8].

Motivated by the shortcomings of traditional approaches, recent work has investigated alternative directory organizations that improve scalability in a single-level directory. Tagless directories [35] use Bloom filters to represent sharer sets. Tagless does not store address tags, making it highly area-efficient (as we will see later, SCD spends more space storing addresses than actual coherence information). Although Tagless reduces area overheads, both area and energy scale linearly with core count, so Tagless is area-efficient but not energy-efficient at 1024 cores [10]. Additionally, it requires significant modifications to the coherence protocol, and incurs additional bandwidth overheads due to false positives. Moreover, Tagless relies on the tracked caches being set-associative, and would not work with other array designs, such as skew-associative caches [27] or zcaches [25]. SPACE [36] observes that applications typically exhibit a limited number of sharing patterns, and introduces a level of indirection to reduce sharing pattern storage: a sharing pattern table encodes a limited number of sharing patterns with full bit-vectors, and an address-indexed sparse directory holds pointers to the pattern table. Due to the limited sharing pattern table size, patterns often need to be merged, and are inexact. However, multiple copies of the sharing pattern table must be maintained in a tiled organization, increasing overheads with the number of tiles [36]. Although these schemes increase the range of sharers that can be tracked efficiently, they are still not scalable and require additional bandwidth.

Alternatively, prior work has proposed coarse-grain coherence tracking [4, 9, 34]. These schemes reduce area overheads, but again increase the number of spurious invalidation and downgrade messages, requiring additional bandwidth and energy. Finally, to reduce directory overheads, WayPoint [18] proposes to cache a sparse full-map directory on the last-level cache, using a hash table organization. This reduces directory overheads and works well if programs have significant locality, but it reduces directory coverage and introduces significant complexity.

As Table 1 shows, all these schemes suffer from one or several significant drawbacks. In contrast, SCD, which we present in this paper, represents sharer sets exactly and in a scalable fashion (both in area and energy), does not require coherence protocol modifications, and can be designed to guarantee an arbitrarily small amount of invalidations. SCD's design relies on the flexibility provided by efficient highly-associative caches, which we review next.

2.2. Efficient Highly-Associative Caches

Recent work has proposed cache designs that provide high associativity with a small number of ways. Both ZCache [25] and Cuckoo Directory [10] build on skew-associative caches [27] and Cuckoo hashing [24]. As shown in Figure 1, these designs *use a different hash function to index each way*, like skew-associative caches. Hits require a single lookup, but on a replacement, these designs leverage the multiple hash functions to provide an arbitrarily large number of replacement candidates in multiple steps, increasing associativity. Both schemes differ in how the replacement process is implemented.

Cuckoo Directory uses W-ary Cuckoo hashing to implement its replacement process: when inserting a new line, if there are no unused entries, the incoming line replaces one of the existing ones, which is taken out and then reinserted in one of the other positions it can map to. This process is repeated until either an unused entry is found, or a threshold of reinsertion attempts is reached. In this case, the line that was last taken out is evicted. Ferdman et al. build sparse directories using this technique and empirically show that it reduces evictions (and therefore directory-induced invalidations) to negligible levels with arrays that are somewhat larger than the caches they are tracking [10].

ZCache implements the replacement process differently: it first retrieves all possible replacement candidates in a breadth-first fashion, selects the least desirable candidate using replacement policy information, and performs a few moves to evict that candidate and insert the new one. This process requires fewer lookups and far fewer moves than Cuckoo Directory (saving energy), can be pipelined (reducing latency), and enables using a replacement policy. For example, in a 4-way array, evaluating 52 replacement candidates requires 13 lookups and at most 2 moves in a zcache, but 17 lookups and 16 moves in a Cuckoo Directory. However, the Cuckoo Directory replacement process can stop early, while zcaches expand a fixed number of candidates. SCD combines the best features from both schemes to implement its replacement process.

Finally, due to the multiple hash functions and replacement process, these arrays can be characterized with simple, workload-independent analytical models that are accurate in practice. Prior work leverages these models to show that associativity depends only on the number of replacement candidates, not ways [25], and to implement scalable and efficient cache partitioning [26]. In this paper, we extend these models

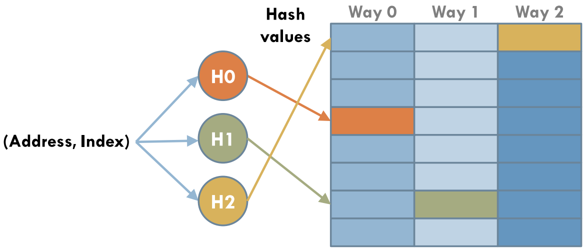


Figure 1: Example 3-way array organization used.

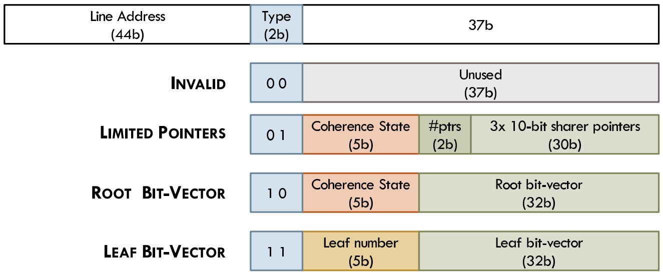
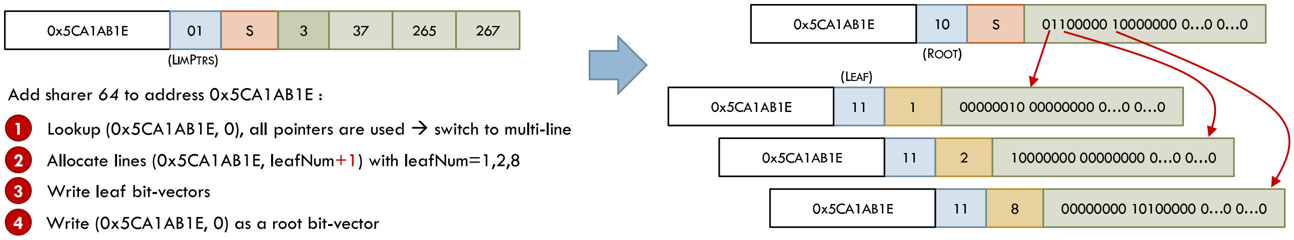


Figure 2: SCD line formats. Field widths assume a 1024-sharer directory.



Add sharer *64* to address 0x5CA1AB1E :

1. Lookup (0x5CA1AB1E, 0), all pointers are used → switch to multi-line
2. Allocate lines (0x5CA1AB1E, leafNum+1) with leafNum=1,2,8
3. Write leaf bit-vectors
4. Write (0x5CA1AB1E, 0) as a root bit-vector

Figure 3: Example operation: adding a sharer to a full limited pointer line.

to characterize and show how to provision directories implemented with these arrays, including SCD and sparse directories, formalizing the empirical results of Ferdman et al. [10].

3. Scalable Coherence Directory

To scale gracefully, SCD exploits the insight that a directory does not need to provide enough capacity to track a specific number of addresses, but a specific number of *sharers*, ideally as many as can fit in the tracked caches. However, sparse directories use address-indexed arrays, so they use one directory tag per address. Instead, SCD represents sharer sets using a variable number of tags per address. Lines with one or a few sharers use a single directory tag with a limited pointer format, and widely shared lines employ a multi-tag format using hierarchical bit-vectors. We first describe the array used to hold the directory tags, then explain how SCD represents and operates on sharer sets.

3.1. SCD Array

Figure 1 shows the structure of the SCD array. It is similar to skew-associative caches, zcaches and Cuckoo Directories, with one hash function per way. However, hash functions take the concatenation of the line address and an *index* as input instead of using just the address. Every line in the directory will have a tag with index 0. Additionally, lines that use more than one directory tag will have those additional tags at locations with indexes other than 0. These indexes need not be consecutive, and are included in the hash functions so that multiple tags representing the same line map to different sets.

SCD's replacement process is very similar to zcache's, since it is more efficient than Cuckoo Directory's, as explained in Section 2.2. However, SCD does not pipeline the replacement process, and stops looking for candidates as soon as an empty tag is found. As we will see, this greatly improves directory efficiency. SCD can optionally implement a replacement policy. However, replacement policies only make sense for underprovisioned directories — in Section 4 we will see that SCD can be sized to cause a negligible amount of evictions regardless of the workload, making a replacement policy unnecessary.

3.2. Line Formats

SCD encodes lines using different tag formats. Figure 2 illustrates these formats for a 1024-sharer directory. Lines with few sharers use a single-tag, *limited pointer* representation, with three pointers in the example. When more sharers are needed, SCD switches to a multi-tag format using hierarchical bit-vectors. In the example, a 32-bit *root* bit-vector tag indicates which subsets of cores share the line, while a set of 32-bit *leaf* bit-vectors encode the sharers in each subset. Leaf bit-vectors include a *leaf number* field that encodes the subset of sharers tracked by each leaf (e.g., leaf number 0 tracks sharers 0–31, 1 tracks 32–63, and so on). We will explain SCD's operation using this two-level representation, but this can be easily extended to additional levels.

3.3. Directory Operations

SCD needs to support three operations on sharer sets: adding a sharer when it requests the line, removing a sharer when it writes back the line, and retrieving all sharers for an invalidation or downgrade.

Adding and removing sharers: On a directory miss, the replacement process allocates one tag for the incoming line with index 0 (possibly evicting another tag). This tag uses the limited pointer format. Further sharers will use additional pointers. When a sharer needs to be added and all the pointers are used, the line is switched to the multi-tag format as follows: First, the necessary bit-vector leaves are allocated for the existing pointers and the new sharer. Leaf tags are then populated with the existing and new sharers. Finally, the limited pointer tag transitions to the root bit-vector format, setting the appropriate bits to 1. Figure 3 illustrates this process. Removing a sharer (due to clean or dirty writebacks) follows the inverse procedure. When a line loses all its sharers, all its directory tags are marked as invalid.

Invalidations and downgrades: Invalidations are caused by both coherence (on a request for exclusive access, the directory needs to invalidate all other copies of the line) and evictions in the directory. Downgrades only happen due to coherence (a request for read on an exclusive line needs to downgrade the exclusive sharer, if any). Coherence-induced invalidations are trivial: all the sharers are sent invalidation messages. If the address is represented in the hierarchical bit-vector format, all leaf bit-vectors are marked as invalid, and the root bit-vector tag transitions to the limited pointer format, which then encodes the index of the requesting core.

In contrast, eviction-induced invalidations happen to a specific *tag*, not an address. Limited pointer and root bit-vector tag evictions are treated like coherence-based invalidations, invalidating all sharers so that the tag can be reused. Leaf bit-vector evictions, however, only invalidate the subset of sharers represented in the tag. As we will see later, eviction-induced invalidations can be made arbitrarily rare.

Additional levels and scalability: SCD can use hierarchical bit-vector representations with more than two levels. A two-level approach scales to 256 sharers with ∼16 bits devoted to track sharers (pointers/bit-vectors) per tag, 1024 sharers with ∼32 bits, and 4096 sharers with ∼64 bits. A three-level representation covers 4096 sharers with ∼16 bits, 32768 sharers with ∼32 bits, and 256K sharers with ∼64 bits. Four-level implementations can reach into the millions of cores. In general, space and energy requirements increase with $O(logN)$, where N is the number of sharers, because the limited bit-vectors and the extended address space increase logarithmically. Since each tag needs on the order of 40-50 bits to store the line address anyway, having on the order of 16-32 bits of sharer information per tag is a reasonable overhead.

3.4. Implementation Details

SCD's multi-tag format achieves the scalability of hierarchical directories, but since *all tags are stored in the same array, it can be made transparent to the coherence protocol*. We now discuss how to implement SCD to make it completely transparent to the protocol, and take the delay of additional accesses out of the critical path, providing the performance of a sparse directory.

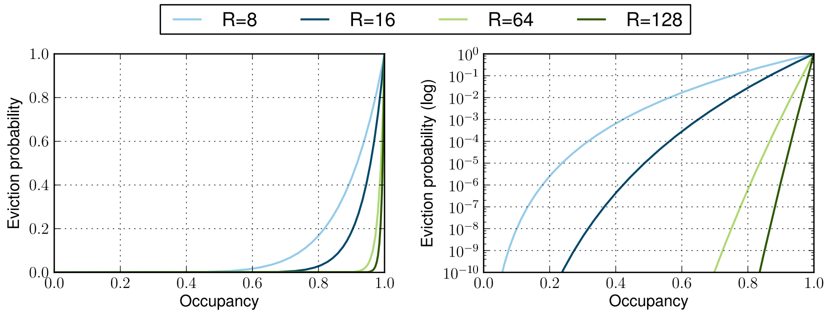
Scheduling: Directories must implement some scheduling logic to make operations appear atomic to the protocol while ensuring forward progress and fairness. This is required in both sparse directories and SCD. For example, in a sparse directory adding a sharer is a read-modify-write operation, and the scheduling logic must prevent any intervening accesses to the tag between the read and the write (e.g., due to a conflicting request or an eviction). However, because SCD operations sometimes span multiple tags, ensuring atomicity is more involved. Note that the access scheduling logic makes the array type transparent to the directory: so long as the SCD array maintains atomicity, forward progress and fairness, it can be used as a drop-in replacement for a sparse array, with no changes to the coherence protocol or controller.

Our SCD implementation satisfies these goals with the following scheduling policies. First, as in conventional sparse directories, concurrent operations to the same address are serialized, and processed in FCFS order, to preserve atomicity and ensure fairness. Second, as in zcaches [25], the array is pipelined, and we allow concurrent non-conflicting lookups and writes, but only allow one replacement at a time. If the replacement process needs to move or evict a tag from an address of a concurrent request, it waits until that request has finished, to preserve atomicity. Third, similar to prior proposals using Cuckoo hashing where insertions are sometimes on the critical path [10, 19], we introduce an *insertion queue* to avoid the latency introduced by the replacement process. Tags are allocated in the insertion queue first, then inserted in the array. We have observed that, in practice, a 4-entry insertion queue suffices to hide replacement delay for sufficiently provisioned directories, where replacements are short, while severely underprovisioned directories require an 8-entry queue. Finally, to avoid deadlock, operations that require allocating new tags and block on a full insertion queue are not started until they allocate their space. This way, the replacement process is able to move or evict tags belonging to the address of the blocking request.

Performance: With this implementation, operations on a specific address are executed atomically once they start. Operations that require allocating one or more tags (adding a sharer) are considered completed once they have reserved enough space in the insertion queue. Writebacks, which require removing a sharer and never allocate, are considered complete once the root tag is accessed. Therefore, *adding and removing a sharer are typically as fast in SCD as in a conventional sparse directory*. On the other hand, coherence-induced invalidations on widely shared addresses need to access several leaf tags to retrieve the sharers, invalidate them, and respond to the original requester once all invalidated sharers have responded. This could take longer with SCD than with a sparse directory, where the whole sharer set can be retrieved in one access (e.g., processing 1024 vs 32 sharers/cycle in our example). However, the critical-path latency of invalidations is determined by serialization latency in the network, as the directory can only inject one invalidation request per cycle, so SCD and sparse full-map directories perform similarly. Invalidation delays have a small performance impact in our simulations (Section 6), but should they become an issue, they can be reduced by processing invalidations in a hierarchical fashion, using multicast networks, or cruise-missile invalidates [2].

3.5. Storage Efficiency

We define *storage efficiency* as the average number of sharers that SCD encodes per tag. Storage efficiency determines how many directory tags are needed, and therefore how to size the directory. When all the lines have a single sharer, SCD has a storage efficiency of 1 sharer/tag. This is a common case (e.g., running a separate workload on each core, or a multithreaded workload where each working set is thread-



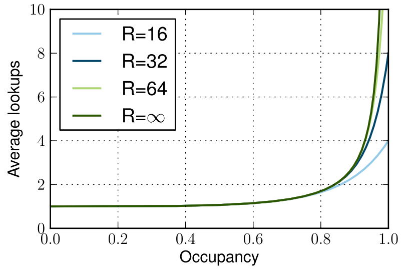


Figure 4: Probability that a replacement results in an eviction as a function of occupancy, for R=8, 16, 64 and 128 replacement candidates, in linear and semi-logarithmic-scales.

Figure 5: Average lookups per replacement as a function of occupancy for a 4-way array.

private and shared code footprint is minimal). When lines have multiple sharers, SCD typically achieves an efficiency higher than 1. For example, using the format in Figure 2, a limited pointer tag with three sharers would have a storage efficiency of 3, while a fully shared line would have an efficiency of 1024 sharers/33 tags $\cong$ 31 sharers/tag. If the expected efficiency is consistently higher than 1, one could undersize or power off part of the directory and still achieve negligible invalidations. Note that SCD has a much lower dynamic range of storage efficiency than sparse directories (1-31 sharers/tag vs 1-1024 sharers/tag) but has far fewer sharer bits per tag (∼32 bits vs ∼1024 bits).

Although, as we will see in Section 6, SCD typically achieves a storage efficiency $\geq$ 1, its *worst-case efficiency* is smaller than 1. In particular, the worst-case efficiency is $1/2$, which happens when a single-sharer line is stored using a two-level bit-vector. Worst-case efficiency decreases as we scale up (e.g., with N-level bit-vectors, it is $1/N$), and might be an issue if the designer wants the directory to provide strict guarantees on evictions (e.g., to avoid interference among applications sharing the CMP). Therefore, we propose two techniques to improve worst-case efficiency.

Line coalescing: A simple optimization is to inspect entries that have few sharers on writebacks, and coalesce them into a limited pointer representation if possible. For example, following Figure 2, if every time we remove a sharer and the root bit-vector has two or fewer bits set, we try to coalesce the line, the worst-case efficiency becomes $2/3$. If we do this with every line with 3 or fewer sharers, the worst-case efficiency becomes $3/4$. Coalescing improves storage efficiency at the expense of additional accesses.

Pointer in root bit-vector: If strict efficiency guarantees are necessary, we can change the tag format to guarantee a worst-case efficiency of 1 by including a single pointer in the root bit-vector tag. When switching a limited pointer tag to a hierarchical bit-vector, the root bit-vector tag keeps one of the sharers in the pointer. If that sharer is removed, one of the sharers represented in the leaf bit-vector tags is moved over to the root tag. With more than two levels, both root and intermediate levels would need to implement the pointer. This guarantees that every tag represents at least one sharer, so the worst-case efficiency is 1. As we will see in Section 4, this enables strict guarantees on directory-induced invalidations. However, this format improves storage efficiency at the expense of additional area. For example, using this format in the example in Figure 2 would require 45 bits/tag for sharer information instead of 39 to hold the extra pointer (assuming we widen the leaf bit-vectors and narrow the root one).

4. Analytical Framework for Directories

We now show that directories built using zcache-like arrays in general, and SCD in particular, can be characterized with analytical models. First, we show that the fraction of replacements that result in evictions and the distribution of lookups per replacement is a function of the directory's occupancy, i.e., the fraction of directory tags used. Second, although directory occupancy is time-varying and workload-dependent, we show that it can be easily bounded. Together, these results show that, with a small amount of overprovisioning, SCD can be designed to guarantee negligible invalidations and high energy efficiency in the worst case.

Uniformity assumption: In our analytical models, we rely on the assumption that the candidates visited in the replacement process have an uniform random distribution over the cache array. We have shown that in practice, this is an accurate assumption for zcaches [25, 26]. We leverage this assumption in the derivations, and verify its accuracy in Section 6 using simulation.

Evictions as a function of occupancy: Assume the directory has T tags, of which U are used. We define directory *occupancy* as $occ = U/T$. Per the uniformity assumption, replacement candidates are independent and uniformly distributed random variables, i.e., $cand_i \sim U[0, T-1]$, and the probability of one being used is $Prob(cand_i\ used) = occ$. If the replacement process, as explained in Section 2.2, is limited to R replacement candidates, the probability that all candidates are being used and we are forced to evict one of them is simply:

$$\begin{aligned} P_{ev}(occ) &= Prob(cand_0\ used \wedge ... \wedge cand_{R-1}\ used) \\ &= Prob(cand_i\ used)^R = occ^R \qquad (1) \end{aligned}$$

Figure 4 plots this probability in linear and semi-logarithmic scales. Note that, with a reasonably large R, the eviction probability quickly becomes negligible. For example, with $R =$

64, $P_{ev}(0.8) = 10^{-6}$, i.e., only one in a million replacements will cause an eviction when the directory is 80% full.

Lookups per replacement: We now derive the distribution and average number of lookups per replacement as a function of occupancy and the number of ways, W. While Equation 1 characterizes worst-case behavior, this illustrates average behavior, and therefore average latency and energy requirements of the directory. First, the probability that all lines are occupied in a single lookup (W ways) is $p = occ^W$. Second, the maximum number of lookups is $L = R/W$. Therefore, the probability of finding an empty line in the k^{th} lookup is $p_k = (1-p)p^{k-1}, k \leq L$. Also, the probability of doing L lookups and not finding any empty line is P_{ev}. Therefore, the average number of lookups is:

$$\begin{aligned} AvgLookups(occ) &= \sum_{k=1}^{L} k(1-p)p^{k-1} + L \cdot P_{ev} \\ &= \frac{1-p^L}{1-p} = \frac{1-occ^R}{1-occ^W} \qquad (2) \end{aligned}$$

Figure 5 plots this value for different numbers of replacement candidates. Fortunately, even for high occupancies, the average number of lookups is much lower than the worst case (R/W). In fact, when evictions are negligible, the average is almost independent of R, and is simply $1/(1-p) = 1/(1-occ^W)$. Therefore, assuming that we design for a negligible number of evictions, the maximum number of candidates R is irrelevant in the average case. In other words, a reasonable design methodology is to first define the target occupancy based on how much extra storage we want to devote versus how expensive allocation operations are, then set R high enough to satisfy a given eviction probability P_{ev}.

Bounding occupancy: Occupancy is trivially bounded by 1.0 (the directory cannot use more lines than it has). However, if we can bound it to a smaller quantity, we can guarantee a worst-case eviction probability and average lookups independently of the workload. In general, the number of used tags is $U = load/eff$, where $load$ is the number of sharers that need to be tracked, and eff is the storage efficiency. Therefore, $occ = U/T = \frac{load/T}{eff}$. We can bound storage efficiency to $eff \geq 1.0$ sharers/line (Section 3.5). With a single-banked directory, the worst-case load is trivially the aggregate capacity of the tracked caches (in lines), which we denote C. Therefore, if we never want to exceed a worst-case occupancy $maxOcc$, we should size the directory with $T = C/maxOcc$ tags. This in turn limits P_{ev} and $AvgLookups$. For example, to ensure that the occupancy never exceeds 90%, we would need to overprovision the directory by 11%, i.e., have 11% more tags than lines are tracked, and with a 4-way, 64-replacement candidate array, this would yield worst-case $P_{ev}(0.9) = 10^{-3}$ and worst-case $AvgLookups(0.9) = 2.9$ lookups/replacement. If we wanted a lower bound on P_{ev} (e.g., to provide stricter non-interference guarantees among competing applications sharing the CMP), we could use $R = 128$, which would give $P_{ev} = 10^{-6}$, and *still require* 2.9 *average lookups*. Furthermore, most applications will not reach this worst-case scenario, and the directory will yield even better behavior. Alternatively, designers can provision the directory for an expected range of occupancies instead of for the worst case, reducing guarantees but saving storage space. In contrast, set-associative directories need to be overprovisioned by 2× or more to reduce evictions, and provide no guarantees [10].

When directories are banked, as it is commonly done with large-scale designs, this bound needs to be relaxed slightly, because the tracked caches will impose a different load on each bank. If a reasonable hash function is used, distributing addresses uniformly across the K directory banks, from basic probability theory, $load$ has a binomial distribution $\sim B(C, 1/K)$, with mean C/K and standard deviation $\sqrt{C/K \cdot (1-1/K)}$. Therefore, the lower the number of banks, the more concentrated these values will be around the mean. In the CMP we study ($C = 2^{21}$ lines, $K = 64$ banks), the standard deviation is only 0.5% of its mean, and it can be assumed that the worst-case $load \cong C/K$ is constant across banks. In general, both P_{ev} and the number of lookups can be treated as functions of random variable C to determine the exact bounds for a given amount of overprovisioning.

In summary, we have seen that SCD can be characterized with analytical models, and can be tightly sized: high occupancies can be achieved with efficient replacements and incurring a negligible amount of evictions. These models apply to SCD and regular sparse directories implemented with arrays where the uniformity assumption holds (skew-associative caches, zcaches or Cuckoo Directories). We will show that these models are accurate in practice using simulation.

5. Experimental Methodology

Modeled system: We perform microarchitectural, execution-driven simulation using an x86-64 simulator based on Pin [23], and model a large-scale CMP with 1024 cores, shown in Figure 6. Table 2 summarizes its main characteristics. Each core is in-order and single-threaded, modeled after Atom [11], and has split 32KB L1 instruction and data caches and a private, inclusive, 128KB L2. All cores share a 256MB L3, which is kept coherent using a MESI coherence protocol. The CMP is divided in 64 tiles, each having 16 cores, a directory and L3 bank, and a memory controller. Both L2 and L3 are 4-way zcaches [25] with 16 and 52 replacement candidates, respectively. Caches and directories use H_3 hash functions, which are simple to implement and work well in practice [5, 25]. Tiles are connected with an 8×8 mesh network-on-chip (NoC) with physical express links.

The system we model is in line with several large-scale CMP proposals, such as Rigel [17, 18] and ATAC [20], and represents a reasonable scale-up of commercial designs like Tilera's Gx-3100 [30], which has 100 cores and 32 MB of distributed, directory-coherent last-level cache that can be globally shared, and is implemented at 40 nm. We estimate that our target CMP should be implementable at 14 or 11 nm. Using McPAT [22], we find that a scaled-down version of this system with 8 tiles and 128 cores would require 420 mm^2 and 115 W at 32 nm. We use the component latencies of this scaled-down CMP in the 1024-core simulations.

Directory implementations: We compare three different directory organizations: sparse, sparse hierarchical (two-level), and SCD. The classic sparse organization has a full-map 1024-bit sharer vector per line. The hierarchical implementation has a distributed first directory level every two tiles,

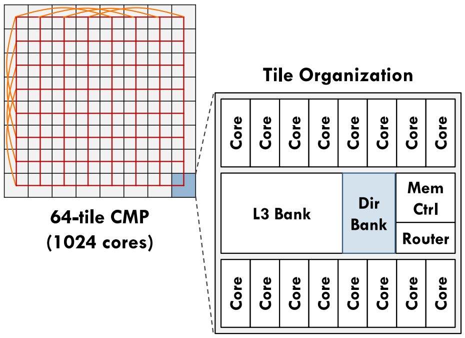


Figure 6: Simulated 64-tile, 1024-core CMP: global tile view (including network links) and tile organization.

Cores	1024 cores, x86-64 ISA, in-order, IPC=1 except on memory accesses, 2 GHz
L1 caches	32 KB, 4-way set associative, split D/I, 1-cycle latency
L2 caches	128 KB private per-core, 4-way 16-candidate zcache, inclusive, 5-cycle latency
L3 cache	256 MB NUCA, 64 banks (1 bank/tile), fully shared, 4-way 52-candidate zcache, non-inclusive, 10-cycle bank latency
Global NoC	8×8 mesh with express physical links every 4 routers, 128-bit flits and links, X-Y routing, 2-cycle router traversal, 1-cycle local links, 3-cycle express links
Coherence protocol	MESI protocol, split request-response, no forwards, no silent drops; sequential consistency
Memory controllers	64 MCUs (1 MCU/tile), 200 cycles zero-load latency, 5 GB/s per controller (optical off-chip interface as in [20])

Table 2: Main characteristics of the simulated 1024-core CMP.

and a second, banked directory level. Therefore, both levels have 32-bit sharer vectors. SCD has the same organization as shown in Figure 2, with 3 limited pointers and a 32/32 2-level bit-vector organization. All organizations nominally use 4-way zcache arrays with 52 replacement candidates and H_3 hash functions, so the sparse organization is similar to a Cuckoo Directory [10]. All directories are modeled with a 5-cycle access latency. We compare directories with different degrees of *coverage*. Following familiar terminology for TLBs, we define coverage as the maximum number of addresses that can be represented in the directory, as a percentage of the total lines in the tracked caches. Therefore, 100%-coverage Sparse and SCD have as many tags as lines in the tracked caches, while a hierarchical directory with 100% coverage has twice as many tags (as each address requires at least two tags, one per level).

Workloads: We simulate 14 multithreaded workloads selected from multiple suites: PARSEC [3] (blackscholes, canneal, fluidanimate), SPLASH-2 [32] (barnes, fft, lu, ocean, radix, water), SPECOMP (applu, equake, wupwise), SPECJBB2005 (specjbb), and BioParallel [16] (svm). We have selected workloads that scale reasonably well to 1024 cores and exhibit varied behaviors in the memory hierarchy (L1, L2 and L3 misses, amount of shared data, distribution of sharers per line, etc.). We simulate complete parallel phases (sequential portions of the workload are fast-forwarded), and report relative execution times as the measure of performance. Runs have at least 200 million cycles and 100 billion instructions, ensuring that all caches are warmed up. We perform enough runs to guarantee stable averages (all results presented have 95% confidence intervals smaller than 1%).

6. Evaluation

6.1. Comparison of Directory Schemes

Directory size and area: Table 3 shows the directory size needed by the different directory organizations (SCD, Sparse, and Hierarchical) for 128 to 1024 cores. We assume line addresses to be 42 bits. Storage is given as a percentage of total tracked cache space. All directories have 100% coverage.

Cores	SCD storage	Sparse storage	Hier. storage	Sparse vs SCD	Hier. vs SCD
128	10.94%	34.18%	21.09%	3.12×	1.93×
256	12.50%	59.18%	24.22%	4.73×	1.94×
512	13.87%	109.18%	26.95%	7.87×	1.94×
1024	15.82%	209.18%	30.86%	13.22×	1.95×

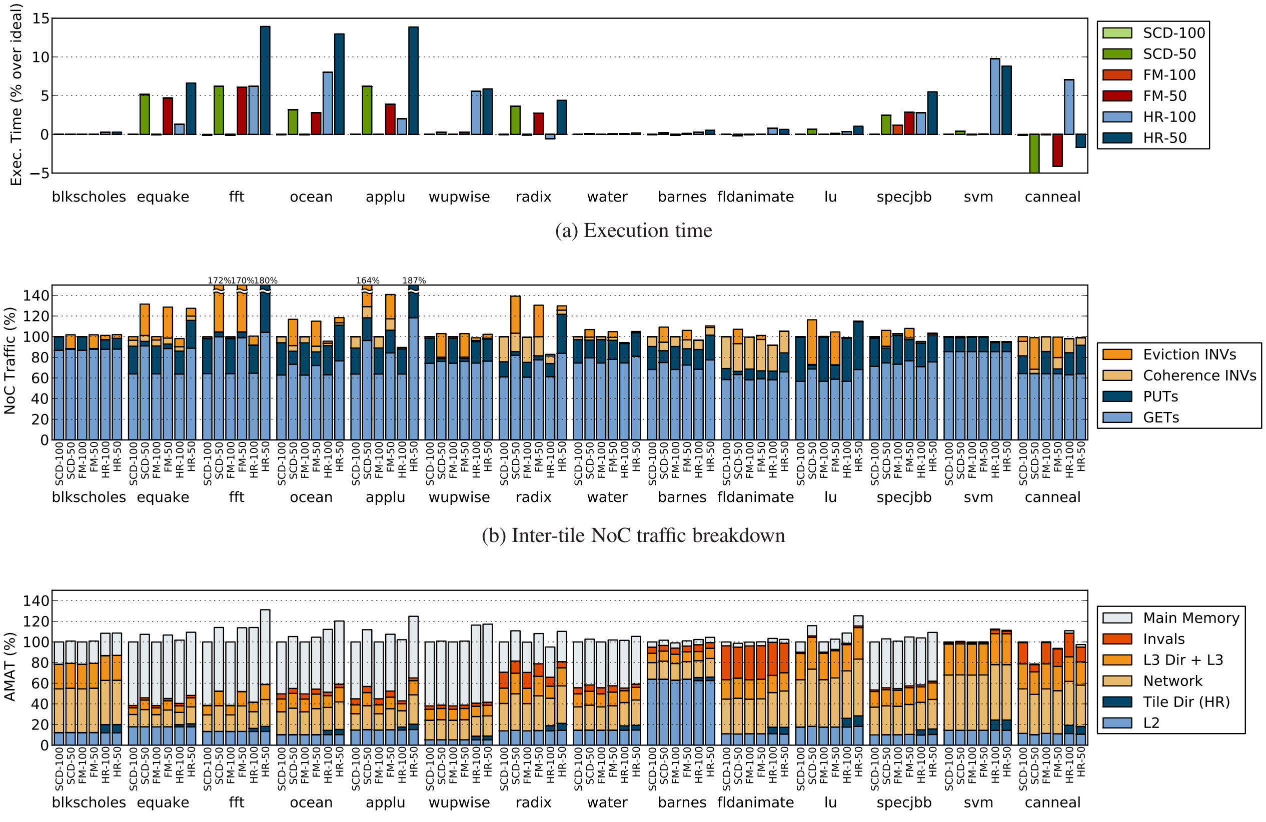
Table 3: Directory size requirements for different organizations. Size is given as a percentage of the aggregate capacity of the tracked caches, assuming a 42-bit line address, 64-byte lines and 100% coverage.

As we can see, SCD significantly reduces directory size. A 2-level SCD uses 3×–13× less space than a conventional sparse directory, and around 2× less than a 2-level hierarchical implementation. A 3-level SCD would be even more efficient (e.g., requiring 18 bits of coherence data per tag instead of 39 at 1024 cores), although gains would be small since the address field would take most of the tag bits.

We can approximate directory area using directory size and assuming the same storage density for the L3 cache and the directory. On our 1024-core CMP, SCD would require 20.2 MB of total storage, taking 3.1% of die area, while a two-level hierarchical directory would require 39.5 MB, taking 6.1% of die area. Sparse directories are basically unimplementable at this scale, requiring 267MB of storage, as much as the L3 cache.

Performance: Figure 7 compares execution time, global NoC traffic and average memory access time among different directory organizations. Each directory is simulated at both 100% and 50% coverage. Smaller values are better for all graphs, and results are normalized to those of an idealized directory (i.e., one with no invalidations). Recall that all directory organizations use 4-way/52-candidate zcache arrays. We will discuss set-associative arrays in Section 6.4.

Looking at Figure 7a, we see that both SCD and Sparse achieve the performance of the ideal directory in all applications when sized for 100% coverage, while their 50%-sized variants degrade performance to varying degrees (except on canneal, which we will discuss later, where performance increases). Underprovisioned Sparse directories perform slightly better than SCD because their occupancy is lower, as they require one line per address. Hierarchical directories, on the other hand, are slightly slower even at 100%



(a) Execution time

(b) Inter-tile NoC traffic breakdown

(c) Average memory access time (AMAT) breakdown

Figure 7: Comparison of nominally provisioned (100% coverage) and underprovisioned (50% coverage) directory organizations: SCD, sparse full-map (FM) and 2-level sparse hierarchical (HR). All directories use 4-way/52-candidate zcache arrays.

coverage, as they require an additional level of lookups, and their performance degrades significantly more in the undersized variant. Note that the 50%-coverage Hierarchical directory has about the same area as the 100%-coverage SCD.

Figures 7b and 7c give more insight into these results. Figure 7b breaks down NoC traffic into GET (exclusive and shared requests for data), PUT (clean and dirty writebacks), coherence INV (invalidation and downgrade traffic needed to maintain coherence), and eviction INV (invalidations due to evictions in the directory). Traffic is measured in flits. We see that all the 100%-sized directories introduce practically no invalidations due to evictions, except SCD on canneal, as canneal pushes SCD occupancy close to 1.0 (this could be solved by overprovisioning slightly, as explained in Section 4). The undersized variants introduce significant invalidations. This often reduces PUT and coherence INV traffic (lines are evicted by the directory before the L2s evict them themselves or other cores request them). However, those evictions cause additional misses, increasing GET traffic. Undersized directories increase traffic by up to 2×. Figure 7c shows the effect that additional invalidations have on average memory access time (AMAT). It shows normalized AMAT for the different directories, broken into time spent in the L2, local directory (for the hierarchical organization), NoC, directory and L3, coherence invalidations, and main memory. Note that the breakdown only shows critical-path delays, e.g., the time spent on invalidations is not the time spent on every invalidation, but the critical-path time that the directory spends on coherence invalidations and downgrades. In general, we see that the network and directory/L3 delays increase, and time spent in invalidations decreases sometimes (e.g., in fluidanimate and canneal). This happens because eviction invalidations (which are not on the critical path) reduce coherence invalidations (on the critical path). This is why canneal performs better with underprovisioned directories: they invalidate lines that are not reused by the current core, but will be read by others (i.e., canneal would perform better with smaller private caches). Dynamic self-invalidation [21] could be used to have L2s invalidate copies early and avoid this issue.

In general, we see that hierarchical directories perform much worse when undersized. This happens because both the level-1 directories and level-2 (global) directory cause invalidations. Evictions in the global directory are especially troublesome, since all the local directories with sharers must be invalidated as well. In contrast, an undersized SCD can prioritize leaf or limited pointer lines over root lines for eviction, avoiding expensive root line evictions.

Energy efficiency: Due to a lack of energy models at 11 nm, we use the number of array operations as a proxy for energy efficiency. Figure 8 shows the number of operations (lookups

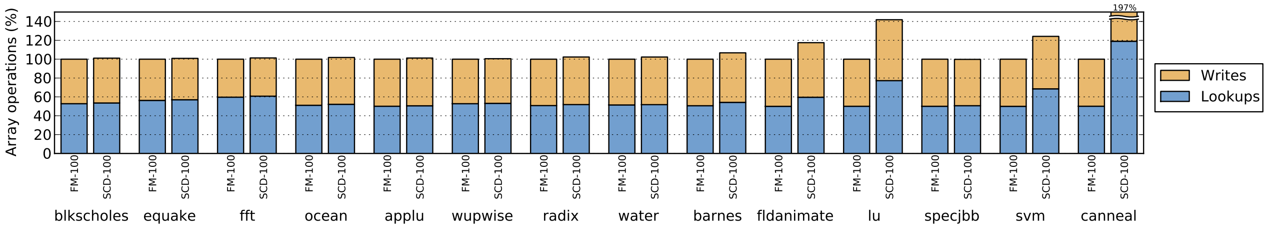


Figure 8: Comparison of array operations (lookups and writes) of sparse full-map (FM) and SCD with 100% coverage.

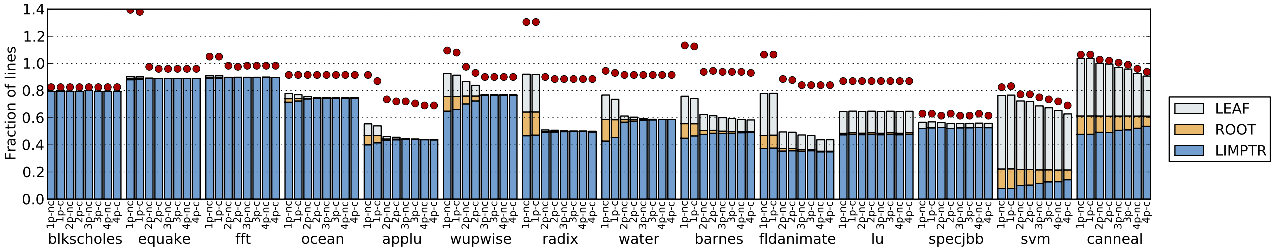


Figure 9: Average and maximum used lines as a fraction of tracked cache space (in lines), measured with an ideal SCD directory with no evictions. Configurations show 1 to 4 limited pointers, without and with coalescing. Each bar is broken into line types (limited pointer, root bit-vector and leaf bit-vector). Each dot shows the maximum instantaneous occupancy seen by any bank.

and writes) done in SCD and Sparse directories. Each bar is normalized to Sparse. Sparse always performs fewer operations because sharer sets are encoded in a single line. However, SCD performs a number of operations comparable to Sparse in 9 of the 14 applications. In these applications, most of the frequently-accessed lines are represented with limited pointer lines. The only applications with significant differences are barnes (5%), svm, fluidanimate (20%), lu (40%) and canneal (97%). These extra operations are due to two factors: first, operations on multi-line addresses are common, and second, SCD has a higher occupancy than Sparse, resulting in more lookups and moves per replacement. However, SCD lines are narrower, so SCD should be more energy-efficient even in these applications.

6.2. SCD Occupancy

Figure 9 shows average and maximum used lines in an ideal SCD (with no evictions), for different SCD configurations: 1 to 4 limited pointers, with and without coalescing. Each bar shows average occupancy, and is broken down into the line formats used (limited pointer, root bit-vector and leaf bit-vector). Results are given as a fraction of tracked cache lines, so, for example, an average of 60% would mean that a 100%-coverage SCD would have a 60% average occupancy assuming negligible evictions. These results show the space required by different applications to have negligible evictions.

In general, we observe that with one pointer per tag, some applications have a significant amount of root tags (which do not encode any sharer), so both average and worst-case occupancy sometimes exceed 1.0×. Worst-case occupancy can go up to 1.4×. However, as we increase the number of pointers, limited pointer tags cover more lines, and root tags decrease quickly (as they are only used for widely shared lines). Average and worst-case occupancy never exceed 1.0× with two or more pointers, showing that SCD's storage efficiency is satisfactory. Coalescing improves average and worst-case occupancy by up to 6%, improving workloads where the set of shared lines changes over time (e.g., water, svm, canneal), but not benchmarks where the set of shared lines is fairly constant (e.g., fluidanimate, lu).

6.3. Validation of Analytical Models

Figure 10 shows the measured fraction of evictions (empirical P_{ev}) as a function of occupancy, on a semi-logarithmic scale, for different workloads. Since most applications exercise a relatively narrow band of occupancies for a specific directory size, to capture a wide range of occupancies, we sweep coverage from 50% to 200%, and plot the average for a specific occupancy over multiple coverages. The dotted line shows the value predicted by the analytical model (Equation 1). We use 4-way arrays with 16, 52 and 104 candidates. As we can see, the theoretical predictions are accurate in practice.

Figure 11 also shows the average number of lookups for the 52-candidate array, sized at both 50% and 100% coverage. Each bar shows the measured lookups, and the red dot shows the value predicted by the analytical model. Again, empirical results match the analytical model. We observe that with a 100% coverage, the number of average lookups is significantly smaller than the maximum ($R/W = 13$ in this case), as occupancy is often in the 70%-95% range. In contrast, the underprovisioned directory is often full or close to full, and the average number of lookups is close to the maximum.

In conclusion, we see that SCD's analytical models are accurate in practice. This lets architects size the directory using simple formulas, and enables providing strict guarantees on directory-induced invalidations and energy efficiency with a small amount of overprovisioning, as explained in Section 4.

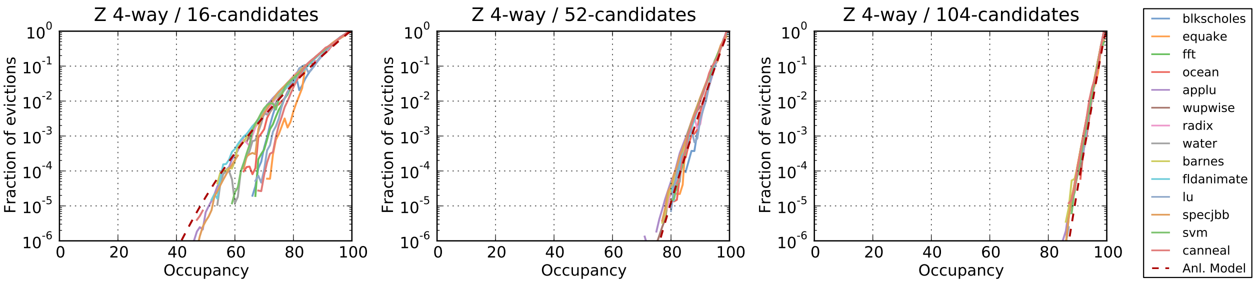


Figure 10: Measured fraction of evictions as a function of occupancy, using SCD on 4-way zcache arrays with 16, 52 and 104 candidates, in semi-logarithmic scale. Empirical results match analytical models.

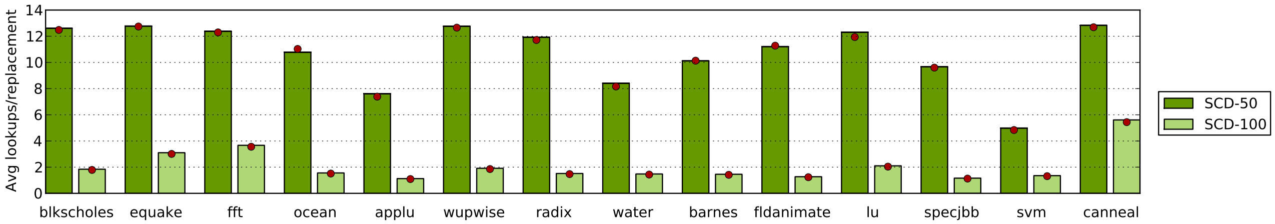


Figure 11: Average lookups per replacement on a 4-way, 52-candidate array at 50% and 100% coverage. Each bar shows measured lookups, and the red dot shows the value predicted by the analytical model. Empirical results match analytical models, and replacements are energy-efficient with sufficiently provisioned directories.

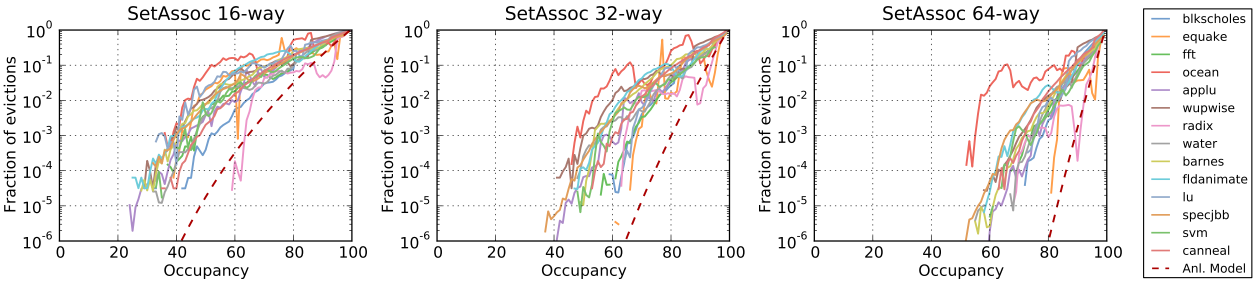


Figure 12: Measured fraction of evictions as a function of occupancy, using SCD on set-associative arrays with 16, 32 and 64 ways, in semi-logarithmic scale.

6.4. Set-Associative Caches

We also investigate using SCD on set-associative arrays. Figure 12 shows the fraction of evictions as a function of occupancy using 16, 32 and 64-way caches. All designs use H_3 hash functions. As we can see, set-associative arrays do not achieve the analytical guarantees that zcaches provide: results are both significantly worse than the model predictions and application-dependent. Set-associative SCDs incur a significant number of invalidations even with a significantly oversized directory. For example, achieving $P_{ev} = 10^{-3}$ on these workloads using a 64-way set-associative design would require overprovisioning the directory by about 2×, while a 4-way/52-candidate zcache SCD needs around 10% overprovisioning. In essence, this happens because set-associative arrays violate the uniformity assumption, leading to worse associativity than zcache arrays with the same candidates.

These findings essentially match those of Ferdman et al. [10] for sparse directories. Though not shown, we have verified that this is not specific to SCD — the same patterns can be observed with sparse and hierarchical directories as well. In conclusion, if designers want to ensure negligible directory-induced invalidations and guarantee performance isolation regardless of the workload, directories should not be built with set-associative arrays. Note that using zcache arrays has more benefits in directories than in caches. In caches, zcaches have the latency and energy efficiency of a low-way cache on hits, but replacements incur similar energy costs as a set-associative cache of similar associativity [25]. In directories, the cost of a replacement is also much smaller since replacements are stopped early.

7. Conclusions

We have presented SCD, a single-level, scalable coherence directory design that is area-efficient, energy-efficient, requires no modifications to existing coherence protocols, represents sharer sets exactly, and incurs a negligible number of invalidations. SCD exploits the insight that directories need to track a fixed number of sharers, not addresses, by representing sharer sets with a variable number of tags: lines with one or few sharers use a single tag, while widely shared lines use additional tags. SCD uses efficient highly-associative caches that allow it to be characterized with simple analytical models, and enables tight sizing and strict probabilistic bounds on evictions and energy consumption. SCD requires 13× less storage than conventional sparse full-map directories at 1024 cores, and is 2× smaller than hierarchical directories while using a simpler coherence protocol. Using simulations of a 1024-core CMP, we have shown that SCD achieves the predicted benefits, and its analytical models on evictions and energy efficiency are accurate in practice.

Acknowledgements

We sincerely thank Christina Delimitrou, Jacob Leverich, David Lo, and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. This work was supported in part by the Stanford Pervasive Parallelism Laboratory. Daniel Sanchez was supported by a Hewlett-Packard Stanford School of Engineering Fellowship.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of the 15th annual Intl. Symp. on Computer Architecture*, 1988.

[2] L. Barroso, K. Gharachorloo, R. McNamara, et al. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proc. of the 27th annual Intl. Symp. on Computer Architecture*, 2000.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. of the 17th intl. conf. on Parallel Architectures and Compilation Techniques*, 2008.

[4] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proc. of the 32nd annual Intl. Symp. on Computer Architecture*, 2005.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proc. of the 9th annual ACM Symposium on Theory of Computing*, 1977.

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

[7] X. Chen, Y. Yang, G. Gopalakrishnan, and C. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer Aided Design*, 2006.

[8] X. Chen, Y. Yang, G. Gopalakrishnan, and C. Chou. Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. *Formal Methods in System Design*, 36(1), 2010.

[9] N. Enright Jerger, L. Peh, and M. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *Proc. of the 41st Annual IEEE/ACM intl. symp. on Microarchitecture*, 2008.

[10] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A scalable directory for many-core systems. In *Proc. of the 17th IEEE intl. symp. on High Performance Computer Architecture*, 2011.

[11] G. Gerosa et al. A sub-1W to 2W low-power IA processor for mobile internet devices and ultra-mobile PCs in 45nm hi-K metal gate CMOS. In *IEEE Intl. Solid-State Circuits Conf.*, 2008.

[12] S. Guo, H. Wang, Y. Xue, C. Li, and D. Wang. Hierarchical Cache Directory for CMP. *Journal of Computer Science and Technology*, 25(2), 2010.

[13] A. Gupta, W. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proc. of the Intl. Conf. on Parallel Processing*, 1990.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (4th ed.).* Morgan Kaufmann, 2007.

[15] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *IEEE Intl. Solid-State Circuits Conf.*, 2010.

[16] A. Jaleel, M. Mattina, and B. Jacob. Last Level Cache (LLC) Performance of Data Mining Workloads On A CMP. In *Proc. of the 12th intl. symp. on High Performance Computer Architecture*, 2006.

[17] J. Kelm, D. Johnson, M. Johnson, et al. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.

[18] J. Kelm, M. Johnson, S. Lumetta, and S. Patel. WayPoint: scaling coherence to 1000-core architectures. In *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010.

[19] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. In *Proc. of the European Symposium on Algorithms*, 2008.

[20] G. Kurian, J. Miller, J. Psota, et al. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010.

[21] A. Lebeck and D. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proc. of the 22nd annual Intl. Symp. in Computer Architecture*, 1995.

[22] S. Li, J. H. Ahn, R. D. Strong, et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009.

[23] C.-K. Luk, R. Cohn, R. Muth, et al. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN conf. on Programming Language Design and Implementation*, 2005.

[24] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proc. of the 9th annual European Symp. on Algorithms*, 2001.

[25] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2010.

[26] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. of the 38th annual Intl. Symp. in Computer Architecture*, 2011.

[27] A. Seznec. A case for two-way skewed-associative caches. In *Proc. of the 20th annual Intl. Symp. on Computer Architecture*, 1993.

[28] J. Shin et al. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *Intl. Solid-State Circuits Conf.*, 2010.

[29] Sun Microsystems. UltraSPARC T2 supplement to the UltraSPARC architecture 2007. Technical report, 2007.

[30] Tilera. TILE-Gx 3000 Series Overview. Technical report, 2011.

[31] D. A. Wallach. PHD: A Hierarchical Cache Coherent Protocol. Technical report, Cambridge, MA, USA, 1992.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995.

[33] Q. Yang, G. Thangadurai, and L. Bhuyan. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3), 1992.

[34] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture*, 2007.

[35] J. Zebchuk, V. Srinivasan, M. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009.

[36] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing pattern-based directory coherence for multicore scalability. In *Proc. of the 19th intl. conf. on Parallel Architectures and Compilation Techniques*, 2010.