Multi-Grain Coherence Directories

Jason Zebchuk
Department of Electrical and Computer Engineering
University of Toronto

Babak Falsafi
EcoCloud, EPFL

Andreas Moshovos
Department of Electrical and Computer Engineering
University of Toronto

ABSTRACT

Conventional directory coherence operates at the finest granularity possible, that of a cache block. While simple, this organization fails to exploit frequent application behavior: at any given point in time, large, continuous chunks of memory are often accessed only by a single core.

We take advantage of this behavior and investigate reducing the coherence directory size by tracking coherence at multiple different granularities. We show that such a Multi-grain Directory (MGD) can significantly reduce the required number of directory entries across a variety of different workloads. Our analysis shows a simple dual-grain directory (DGD) obtains the majority of the benefit while tracking individual cache blocks and coarse-grain regions of 1KB to 8KB. We propose a practical DGD design that is transparent to software, requires no changes to the coherence protocol, and has no unnecessary bandwidth overhead. This design can reduce the coherence directory size by 41% to 66% with no statistically significant performance loss.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms

Design,Performance

Keywords

Cache Coherence, Coherence Directory

1. INTRODUCTION

As chips incorporate ever more cores, coherence directories are increasingly used to maintain coherence among on-chip caches [10, 1, 2, 26, 29]. Compared to other options, directory-based coherence generally uses less network bandwidth and can easily adapt to arbitrary on-chip network topologies. These benefits offer scalable coherence solutions that enable future processors with tens to hundreds of cores to maintain the convenience and compatibility of coherent shared memory [20]. However, even scalable directory designs can incur significant area and energy overheads, especially when implemented in performance-per-watt optimized many-core processors where every square millimetre and millijoule counts. Existing proposals might scale well, but they cannot be shrunk beyond certain limits without sacrificing performance. This work explores alternative directory designs that can push beyond these limitations while maintaining performance.

Traditionally, multiprocessors use one of three basic coherence directory types. The size of *Duplicate Tag* directories scales well, but their highly associative structures result in poor latency and low energy efficiency, making them difficult to implement with many cores [5, 14, 32]. *In-cache* directories are energy efficient and convenient for systems with shared caches, but Martin *et al.* [20] and Zebchuk *et al.* [32] have demonstrated that *Sparse* directories can offer similar benefits with less area and greater flexibility. *Sparse*, or tagged, directory designs sacrifice area scalability for energy efficiency and speed by using low-associative structures where each entry represents the sharing pattern for one block [16]. The choice of sharing pattern representation involves a trade-off between precision and scalability, with less precise representations increasing bandwidth and lowering performance. Additionally, limited associativity causes conflict misses that forcibly invalidate cached blocks. Over-provisioning reduces this problem but increases area.

Past work has addressed many shortcomings of sparse directories; nonetheless, sparse directories still require significant on-chip resources, and practical concerns impose lower limits on their size [20]. Alternate sharing pattern representations [3, 9, 16, 24, 33] and hierarchical approaches [15, 24, 31, 27] reduce the size of individual entries and improve scalability. Other approaches, such as Cuckoo directories [12], and SCD [24] limit the need for over-provisioning and avoid excessive forced invalidations. But while Cuckoo directories and SCD significantly improve the area efficiency of sparse directories, the worst-case scenario still requires them to have at least a 1:1 ratio between the number of directory entries and the number of blocks in the private caches. Unfortunately, near worst-case conditions do happen. Some workloads, especially multi-programmed ones, regularly create scenarios where every core caches a set of mostly unique blocks. This work explores the potential to reduce the directory size below the 1:1 ratio while still

MICRO'46 December 7-11, 2013, Davis, CA, USA.

ACM 978-1-4503-2638-4/13/12 ...$15.00.
http://dx.doi.org/10.1145/2540708.2540739

effectively handling worst-case behavior. The technique presented borrows ideas from SCD and Cuckoo directories, and it can be combined with SCD, as Section 3.4.5 discusses.

This work's motivation is that directories only need to capture a snapshot of sharing behavior at each point in time. While many cores may access a block at different times, at any given moment that block might only by cached by a single core. Thus, nominally shared blocks are often *temporarily* private. Further, many applications exhibit a high degree of spatial locality that extends to their sharing behavior, resulting in large, contiguous, temporarily private memory regions [4]. Tracking coherence for such regions should require storing much less information than tracking arbitrary sharing patterns for each block. Even when every block in the private caches is unique, it is highly unlikely that no spatial locality exists, i.e., every block belongs to a unique memory region.

Alisafaee recently proposed Spatiotemporal Coherence Tracking (SCT) as a mechanism for exploiting temporarily private memory regions [4]. While SCT reduces the directory size, its approach suffers from several shortcomings: *(i)* new race conditions complicate the coherence protocol and increase verification costs; *(ii)* unnecssary speculative snoop messages waste bandwidth; *(iii)* the structure and operation are not clearly defined; *(iv)* energy consumption increases; and *(v)* performance suffers and lacks robustness, with slowdowns up to 16%. Sections 4 and 5 discuss these issues further.

This work takes a new, systematic approach to exploiting the phenomenon of temporarily private regions. First, we describe a conceptual multi-grain directory (MGD) – an idealized directory that dynamically refines the coherence granularity and tracks the largest private regions possible. Our analysis of MGD finds that most of the benefit comes with only two granularities: fine-grain cache blocks, and coarse-grain regions between 1 kB and 4 kB in size. Following this analysis, we propose a practical dual-grain coherence directory design (DGD), and evaluate its performance across several workloads with different program behaviors. While DGD might superficially appear to resemble SCT, substantive differences make DGD consistently better in terms of performance and energy efficiency.

In summary, this work makes the following contributions:

- It analyzes applications' transient sharing behavior at a range of granularities, and demonstrates the potential reduction in the number of entries with a multi-grain coherence directory.
- It shows a dual-grain directory likely offers the best tradeoff between reduced size and complexity, and regions between 1 kB and 4 kB provide the most benefit across a range of workloads.
- It demonstrates a practical DGD design that: *a*) is entirely transparent to software; *b*) requires no coherence protocol changes; *c*) requires no unnecessary snoops to search for cache blocks; *d*) introduces no additional delays on the critical path; *e*) adapts storage usage based on workload behavior; *f*) reduces the directory area by 41% with minimal performance loss; and *g*) reduces the combined L2, L3 and directory energy per instruction by 1%.

Section 2 describes and analyzes the idealized MGD. Section 3 presents a practical DGD design, which Section 4 then compares to the previously proposed SCT. Section 5 evaluates DGD. Section 6 outlines related work. Finally, Section 7 summarizes this work's contributions.

2. IDEAL MULTI-GRAIN DIRECTORY

An *ideal* multi-grain directory (*i*MGD) attempts to dynamically refine the coherence granularity to track as many individual cache blocks with as few directory entries as possible. This approach exploits common program behavior that results in large continuous memory regions being accessed by only one core at any given time. Such regions include truly private data that is exclusively accessed by one core, and a larger class of data that is nominally shared but temporarily private (i.e., accessed by just one core for a period of time, although other cores access it at other times). For each temporarily private region, *i*MGD uses only a single entry instead of using one entry for each block in the region, significantly reducing the number of directory entries compared to traditional designs. In the extreme case of completely segregated memory spaces, each core might require just one directory entry.

Each *i*MGD entry tracks either a temporarily private memory region, or a single cache block with any number of sharers. The size of each region is dynamically adjusted to minimize the number of directory entries. To simplify the conceptual design, regions are restricted to be aligned and to have a size that is a power of two between 128 B (two cache blocks) and 1 GB (sharing behavior mostly flattens out for larger regions). While exploring this idealized design, we ignore details about the format and size of directory entries and focus our exploration on the total number of entries.

*i*MGD aggressively merges, splits, and eliminates directory entries to minimize their total number. Practical implementation considerations may favor a design that is less aggressive; or, conversely, actual implementation details might enable additional optimizations. Regardless, the role of *i*MGD is to assess the potential benefits of adaptively refining the directory's coherence granularity. As a result, this section focuses on the key aspects of this concept and purposefully delays consideration of practical implementation details.

2.1 iMGD Operation

On an initial access, *i*MGD assigns a region with the coarsest possible granularity. Thus, the first access creates an entry for the 1 GB region containing the requested block and assigns that region to the requesting core. Subsequent

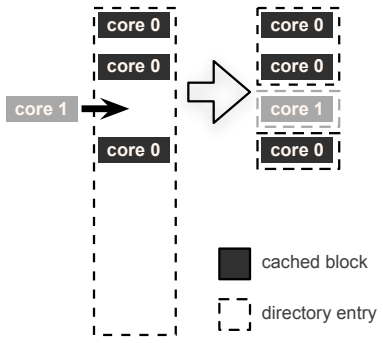


Figure 1: Example of splitting an *i*MGD directory entry.

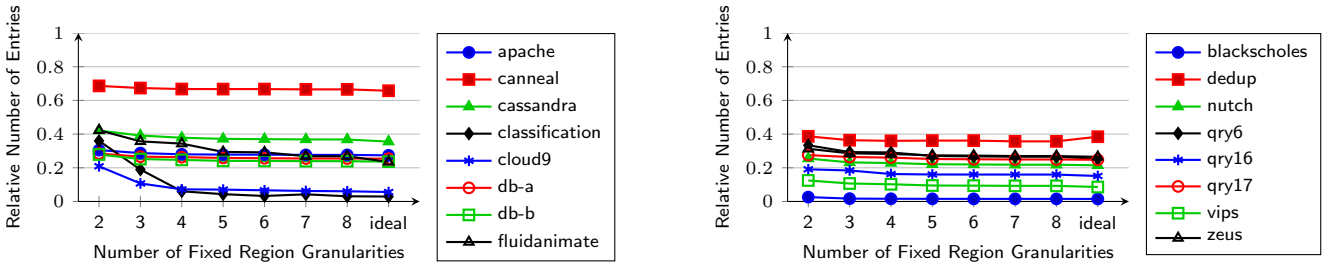


Figure 2: Average entry count for *i*MGD relative to SPARSE$_{1\times}$.

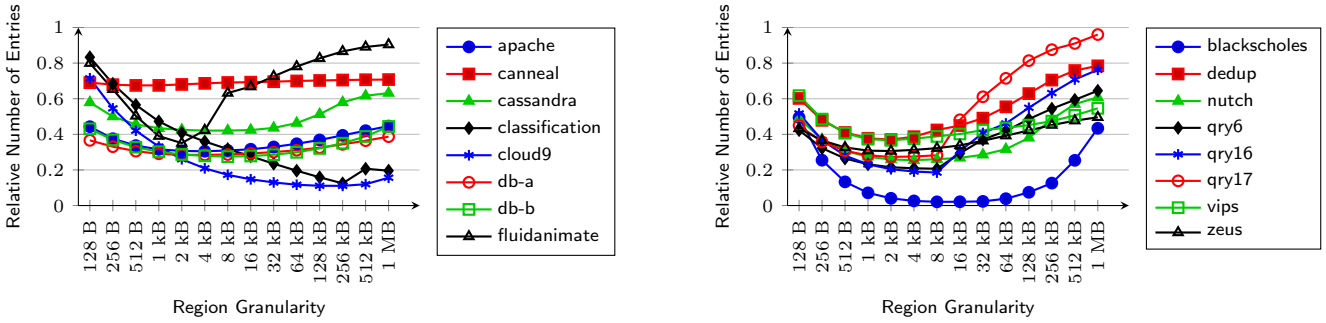


Figure 3: Average entry count for *i*DGD relative to SPARSE$_{1\times}$.

accesses from the same core to the blocks in the same region reuse the same directory entry. If another core accesses this region, *i*MGD splits the entry into smaller regions such that each entry represents either a private region or an individual cache block. Entries are only created for those regions with blocks present in the private caches. Figure 1 illustrates an example of this process where, initially, only core 0 has blocks in some region. A subsequent request from core 1 results in splitting the directory entry into three new entries with finer granularity, while omitting entries for the empty half of the region.

When a core evicts a block from its private cache, *i*MGD attempts to merge entries to create the largest possible private region. Following the example in Figure 1, when the reverse scenario happens and core 1 evicts the cache block, *i*MGD will combine the two directory entries for the remaining blocks cached by core 0 and recreate the original entry for the large region of memory now that is temporarily private again.

2.2 Entry Count Reduction with iMGD

To evaluate *i*MGD's potential, we model a 16-core CMP with 256 kB private L2 caches similar to the Nehalem architecture [1]. Section 5.1 details our methodology. In addition to *i*MGD, we considered constrained MGD designs with two to eight fixed granularities. We chose granularities that optimized behavior across all workloads based on an analysis of *i*MGD using all possible region granularities. Since all workloads use the same fixed granularities, some designs may not be optimal for every workload.

Figure 2 shows the relative entry counts for these MGD designs. The x-axis indicates the number of fixed granularities for each constrained MGD (the *ideal* design has 24 granularities).[1] The results are presented in Figures 2a and 2a. Each graph reports the the number of directory entries relative to the number of unique blocks in the private caches. A ratio of one is equivalent to the number of entries in a duplicate tag directory or an aggressive sparse directory design (SPARSE$_{1\times}$), and comparable to the number of entries in SCD or Cuckoo directories. While these results focus on the *average* number of entries, practical implementations have to either provide more storage to support the worst case, or perform some forced evictions to maintain a smaller size. Section 5 demonstrates practical designs can significantly reduce directory size without sacrificing performance.

Figure 2 indicates *i*MGD can reduce the number of directory entries by more than 60% for all workloads except canneal and by up over 90% for blackscholes, classification, cloud9, and vips. Even in the worst case of canneal the reduction is at least 34%. Canneal implements simulated annealing and has a very irregular access pattern and low spatial locality.

While these results show significant potential for reducing the number of directory entries, the unrestricted *i*MGD design surprisingly offers little benefit over constrained

[1]The region sizes range from individual 64 B cache blocks to 1 GB regions. For each design, we use a single set of region sizes that minimizes the average number of entries across all workloads. The specific combinations are: 2: {64 B, 4 kB}, 3: {64 B, 1 kB, 64 kB}, 4: {64 B, 1 kB, 32 kB, 1 MB}, 5: {64 B, 512 B, 4 kB, 64 kB, 2 MB}, 6: {64 B, 512 B, 4 kB, 32 kB, 256 kB, 4 MB}, 7: {64 B, 512 B, 2 kB, 16 kB, 128 kB, 1 MB, 8 MB}, 8: {64 B, 512 B, 2 kB, 16 kB, 128 kB, 1 MB, 4 MB, 32 MB}.

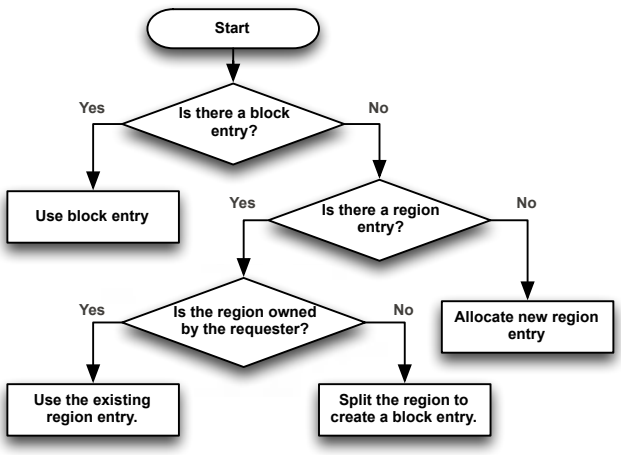


Figure 4: Typical DGD access.

designs. Overall, two fixed granularities offers nearly the same potential benefit as continuously adapting between 24 different granularities. Intuitively, two observations can help explain these results: (1) Common programming and operating system techniques, such as the use of pages, naturally divide memory into fixed granularities. While virtual addresses are contiguous within a physical page, adjacent virtual pages are not necessarily contiguous within physical memory. This behavior can create a "natural" granularity for temporarily private regions while artificially reducing opportunities for larger granularities. (2) For designs used in Figure 2, the smallest region larger than an individual cache block always contains at least eight cache blocks. Thus, the first level alone is capable of reducing the number of directory entries by up to 87.5% in an ideal scenario. Instances of significant spatial locality obtain large benefits from the first few granularities, and additional granularities provide diminishing returns.

Figure 3 further analyzes the behavior of ideal dual-grain directory (*i*DGD) designs which track individual blocks and temporarily private regions of one fixed size. Region sizes range from 128 B (two blocks) up to 128 MB(two million blocks). Most workloads see significant improvement as the region increases up to 1 kB. Behavior then usually stabilizes until extremely large regions eventually exceed the size of the applications temporarily private regions, causing *i*DGD to track more individual cache blocks.

As mentioned previously, the page size makes a natural choice for the region size, and Figure 3 shows that most workloads perform well with 8 kB regions that match our processor's page size. However several workloads, particularly fluidanimate, work better with smaller region sizes. Conversely, classification can take advantage of regions significantly larger than 8 kB, benefiting from its use of super-pages. Overall, regions between 1 kB and 4 kB result in the fewest directory entries, and any practical dual-grain directory design should obtain the majority of the benefit with any of these sizes.

3. DESIGN OF A PRACTICAL DUAL-GRAIN DIRECTORY

This section describes a practical dual-grain directory (DGD) implementation that can achieve the potential area savings suggested by the previous section. In addition to reducing the directory size, our DGD design has the following properties: *1*) Each access requires only a single lookup. *2*) One structure holds both block and region entries in any combination. *3*) Low-associativity reduces lookup costs, while skew-associativity and Zcache-style replacement reduce conflict misses. *4*) No coherence protocol changes are required. *5*) Software transparency for applications and the OS.

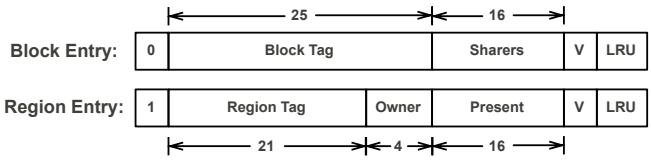


Figure 5: Formats of block and region entries in DGD.

Similar to the conceptual *i*DGD, the practical DGD structure contains a pool of entries, each representing either a single cache block or a coarse grain memory region. A typical directory request will proceed as Figure 4 shows. In the common case, a request will find and use a matching block or region entry for the same core. If no matching entry is found, one is created for the region. Finally, requests that find a region entry from another core will split it and create a new entry for the requested block. Beyond this basic operation there are several important questions to address:

- ⇒ What is the format of a directory entry (Sec. 3.1)?
- ⇒ How are entries located (Sec. 3.2.1) and allocated (Sec. 3.2.2) in the directory structure?
- ⇒ How are entries updated (Sec. 3.2.1 and Sec. 3.4)?
- ⇒ What happens when a region entry is evicted (Sec. 3.2.3)?
- ⇒ How are region entries split into block entries (Sec. 3.3)?
- ⇒ How are block entries fused into region entries (Sec. 3.3)?

The proposed DGD design balances the conflicting goals of reducing area and latency while maintaining simplicity and efficiency. The rest of this section describes the structure and operation of DGD and addresses the above questions.

3.1 DgD Entry Format and Use

Figure 5 shows DGD block and region entry formats. The first bit indicates whether it is a block or region entry. Both entry types contain a tag to identify the region or block address, a valid bit (V), and replacement policy bits (indicated as LRU bits, but other policies are possible). Block entries store a sharing vector with one bit per core. Region entries contain two unique fields: (1) the *owner* identifies the private cache currently caching blocks from the region, and (2) the *present* vector indicates which blocks the owner has cached. The figure shows an example for a 16-core CMP where block entries have a 16-bit sharing vector, and regions are 1 kB and contain sixteen 64 B blocks. As Figure 5 shows, both entry types have the same size. For 16 to 64 cores, region and block entries can be kept the same size while using regions between 1 kB and 4 kB (see results in Section 2.2). In these cases, the owner field in the region entry can be accommodated by the reduced length

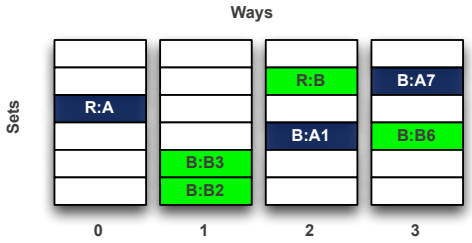


Figure 6: Simultaneous lookup of region and block entries.

of the region tag compared to the block tag, using an extra bit from the sharer field when the number of cores is not a power of two.

3.2 Allocating and Locating Entries

DGD provides fast, efficient lookups while making effective use of limited storage capacity. It uses a low-associative structure that requires just one lookup to find sharing information for any block. Moreover, replacements benefit from the equivalent of a highly associative structure to reduce conflict misses and increase utilization.

A single set-associative structure stores block and region entries. Each directory way uses a different hash function, similar to a skew-associative cache [25]. The first bit of the entry (Figure 5) is concatenated with either the block tag or the region tag (zero padded) to produce the input to each set index hash function.

3.2.1 Entry Lookup

A request might require information from either a block or a region entry. A simple design would first search for a block and then a region entry (Section 3.3 explains why this order cannot be reversed). However, such serialization would add delay to the many requests requiring two accesses. DGD avoids these delays by carefully mapping block entries and corresponding region entries to different ways and searching for both entry types in parallel.

On each access, DGD searches half of the ways for a region entry and half for a block entry. The region tag is used as the input to a hash function that selects which ways search for a block and which for a region. Thus, for a given region, the region entry is restricted to half of the ways, and the block entries are restricted to the other half. However, across all regions, the block and region entries are distributed across all ways and any entry can track either a block or region.

With this organization, one lookup suffices to find either a block, or region entry, or both (Section 3.3 explains when both entry types might co-exist for a given request). Figure 6 shows an example layout with four ways, in which the entry for region A (R:A) is located in way 0, while entries for two blocks in region A (B:A1 and B:A7) are mapped to ways 2 and 3. For region B, on the other hand, the block entries are mapped to ways 1 and 3, while the region entry is allocated in way 2 (but could also have been allocated in way 0).

Splitting the ways between regions and blocks for each region effectively reduces lookup associativity. However, the replacement policy described next mitigates this by effectively increasing the associativity when performing replacements. The results in Section 5 indicate DGD performs well with an associativity of eight.

3.2.2 Replacement Policy

When replacing an entry, DGD takes advantage of the different, per way hash functions and tries to *shuffle* the existing entries to accommodate the allocation without evicting any valid entry. Similar to Zcache [23] and SCD [24], DGD performs a breadth-first search of possible replacement candidates looking for an invalid entry. The search terminates when an available slot is located, or when a limit has been reached for the number of lookups to perform. If no invalid entries were found, DGD uses a global timestamp replacement policy – each block stores a 4-bit timestamp and the oldest block is selected as a victim [23]. Block and region entries are selected with equal preference. Once a victim is selected, entries are moved within the structure to replace the victim and make a place available for the newly allocated entry. This replacement process can require many directory lookups, so an insertion buffer is used to delay the replacement process and remove it from the critical path.

3.2.3 Back-Invalidation of Cached Blocks

When DGD evicts an entry, any cache blocks it represents are invalidated. For block entries, invalidations are multicast to all sharers identified in the sharing vector. For region entries, invalidations are sent to the region owner for each block indicated in the present vector. This process could be optimized with a special region invalidate message that sends the present vector to the cache, but we do not explore this option. Directory evictions are buffered to remove them from the critical path.

3.3 Splitting and Merging Region Entries

While a region is temporarily private and cached by a single owner, all blocks in that region can be represented by one DGD entry. When a request for a block in such a region arrives from a core other than the owner, the region becomes shared and needs to be split. DGD creates a new entry for the block and marks the requesting node as a sharer. If the region owner already has a copy of the block, then the owner is also added to the sharing vector, and the corresponding bit in the present vector is cleared.

Once a block entry has been split from a region entry, the region is no longer temporarily private. However, the region entry can still persist in the directory. Additional block entries can be created on-demand as blocks are accessed by cores other than the owner. Subsequent requests for these blocks are guaranteed to find and use the block entries since DGD always searches for block and region entries in parallel. [2] Thus, even though the region is no longer strictly private, it can still be used to track individual blocks that remain private to the region owner. When the region owner has no more private blocks, DGD removes the region entry.

By letting region entries persist after being split, DGD differs significantly from the *i*MGD concept. For the example in Figure 1, *i*MGD splits a single region entry into three smaller entries when a second core accessed a block in the private region. In contrast, DGD keeps the existing region entry owned by core 0, and only creates one new block entry for the incoming block accessed by core 1. This simplifies the process of splitting entries compared to *i*MGD, and reduces how many entries need to be stored.

[2]This is why serial lookups would always search for a block entry before searching for a region entry.

Whenever a region entry is evicted, or when a sharer is removed from a block entry, it might be possible to merge multiple block entries into a region entry. The *i*MgD design of Section 2 aggressively takes advantage of all such opportunities, but for DgD this would require scanning for all possible block entries within the same region. This could also result in thrashing behavior if shared blocks frequently move back and forth between cores. To avoid such overheads, DgD does not aggressively merge blocks into region entries. Instead DgD only attempts to merge entries when evicting a block entry with a single sharer. In this scenario, DgD searches for a region entry, and if the block sharer owns the region, then DgD merges the block back into the region, creating space in the directory without evicting any cached blocks.

3.4 Putting it All Together

To clarify DgD's operation, this section walks through the possible scenarios for a read request that arrives at the DgD directory. Other requests, e.g. writes, follow a similar sequence of actions. First, the directory performs a single lookup, searching half of the ways for a block entry, and the other half for a region entry. Once this lookup completes, there are four possible scenarios:

1. **No block or region entry exist.** DgD creates a new *region* entry, sets the owner, and marks the requested block in the present vector. The block cannot have any sharers so it is retrieved from either the shared cache or from memory. No special actions are required to account for block entries that might exist for other blocks within the region, and no present bits are set for such blocks, as any accesses to them will always use the block entry, as item 3 describes.

2. **Only a block entry exists.** The sharing vector provides information about all the sharers for this block. A read request can either be satisfied from the shared cache, or from one of the sharers. A write request invalidates the other sharers. In both cases, the sharing vector is updated to indicate the new set of sharers.

3. **Both a block and region entry exist.** The block entry takes precedence, and DgD behaves as if only the block entry exists. The region entry does not track the requested block. This simplifies the required logic and helps avoids potential thrashing when actively shared blocks become temporarily private.

4. **Only a region entry exists.** Two scenarios can occur:

   i) The requester is the region owner. The region is still private, so the block can be retrieved from memory or the shared cache, and the block is then added in the region's present vector.

   ii) The requester is not the region owner. A new *block* entry is created. If the block is marked present in the region entry, then this bit is cleared in the present vector and the owner is added to the sharing vector of the new block entry. Once the new entry has been created, operation proceeds the same as for other cases that use a block entry.

All private caches notify the directory on block replacements to keep the directory up to date. Block entries in the directory are marked invalid when they no longer have any sharers. Similarly, region entries are marked invalid when the owner no longer has any blocks cached from the region.

3.4.1 Effects on the Coherence Protocol

DgD does not change the coherence protocol – it only requires the common features of clean evict notifications and a sharing vector in the directory. DgD does not add any unnecessary snoops to check if blocks in a region might be cached by the owner. The present vector always contains precise information without the need for extra snoops. DgD does not introduce any delays on the critical path. A single lookup locates block and region entries in parallel, and an insertion queue removes long-latency insertions from the critical path.

DgD can operate correctly with alternative sharing representations, and without clean evict notifications. Such changes might increase bandwidth or forced invalidations.

3.4.2 Deadlock and Starvation

DgD reserves insertion queue slots before initiating new requests to avoid deadlock scenarios. The queue allows atomic insertions and replacements off the critical path without introducing race conditions. Further, DgD requires no coherence protocol changes that might introduce new deadlock, livelock or race conditions.

3.4.3 Effect of Forced Evictions

DgD markedly reduces directory size. Effective DgDs have fewer directory entries than there are blocks in the private caches. For workloads with pervasive region sharing or low spatial locality, conflict misses in the directory forcibly evict cached blocks. However, our results demonstrate that temporarily private regions are common in almost all parallel applications and the frequency of forced evictions is low for practically sized DgDs.

3.4.4 Multi-Programmed Workloads

While this work does not study multi-programmed workloads, we expect DgD to excel in this environment, since it should naturally result in many temporarily private regions. Thus, DgD should be able to reduce directory storage requirements without sacrificing performance not only for parallel workloads, but also for multi-programmed workloads. Demonstrating DgD's utility for multi-programmed workloads is left for future work.

3.4.5 Scaling DgD to Hundreds of Cores

DgD's scalability is limited by the block entries' sharing vectors. However, DgD can easily be extended to use other sharing pattern representations to reduce the size of block entries for large core counts. One interesting approach would combine SCD's hierarchical entries [24] with DgD's dual-grain approach. The resulting structure would use present vectors for region entries, and block entries would use SCD's various hierarchical formats. Cache blocks with fewer entries would use one directory block entry, and widely shared blocks would use multiple block entries to track all sharers. The resulting directory structure could easily scale to hundreds or thousands of sharers, and the combined approach would likely require much less area than

the original SCD design.

3.4.6 Tile Interleaving

A distributed DgD directory works best when the tile interleaving is the same as the region size. However, DgD can use non-contiguous regions to allow more fine grain interleaving. Section 5 shows that a 1 kB interleaving for DgD does not cause a significant performance loss compared to the 64 B interleaving used with a sparse directory; thus we do not explore alternative interleavings in this work.

4. DGD VS. SCT

Alisafaee has proposed Spatiotemporal Coherence Tracking (SCT) which follows a similar high-level intuition as DgD [4]. Section 2 provides unique evidence that the dual-grain approach used by both DgD and SCT offers the best trade-off between complexity and reduced directory size. However, SCT has a number of deficiencies. It uses an imprecise representation that causes *1*) increased coherence traffic, *2*) directory and cache scanning, and *3*) new protocol races; and it also effectively couples block and region entries, increasing contention and energy consumption.

SCT's single directory structure contains both block and region entries. Block entries use the same format as DgD (Figure 5). Region entries contain a region tag, a region owner, and two counters. The *private block counter* (PBC) tracks the number of private blocks cached by the region owner, and the *shared block counter* (SBC) tracks the number of SCT block entries within the region. The first access to a region creates a new region entry belonging to the requester. Subsequent requests within the region from the owner increment the PBC, and requests from other cores allocate block entries and increment the SBC.

Many of SCT shortcomings result from its use of counters. The SBC and PBC are inherently *imprecise* – they indicate only the number of blocks without specifying which blocks. The following paragraphs outline the problems caused by this imprecision.

Unnecessary Snoops and Scanning: When the private block counter for a region is non-zero, the owner might cache any block in the region, and requests from other cores often need to speculatively probe the region owner's cache. Also, when evicting a region entry, SCT must scan the directory to find all block entries counted by the SBC, and scan the region owner's cache to find all blocks counted by the PBC. This changes the coherence protocol, wastes network bandwidth, and increases contention and energy consumption due to extra lookups.

Protocol Races: Perhaps more importantly, the SCT's imprecision causes new races that must be properly handled by the coherence protocol. For example, in Figure 7 the directory evicts a block entry and invalidates its sharers, while at the same time the region owner tries to evict the same block. If the evict notification arrives at the SCT directory after the block entry has been removed, it appears to for a private region block, and a naïve implementation would decrement the PBC counter at this point, resulting in an inconsistent state. This is just one example of a potential new race introduced by SCT. SCT might introduce other race conditions, and it is not clear how easy they might be to handle, or how difficult the protocol might be to verify.

Coupling: SCT's shared block counter creates a coupling between region and block entries. Accesses that create or remove a block entry must also update the region entry, increasing energy consumption in the directory. Further, a region entry must exist for every block entry, increasing contention and conflict misses in the directory.

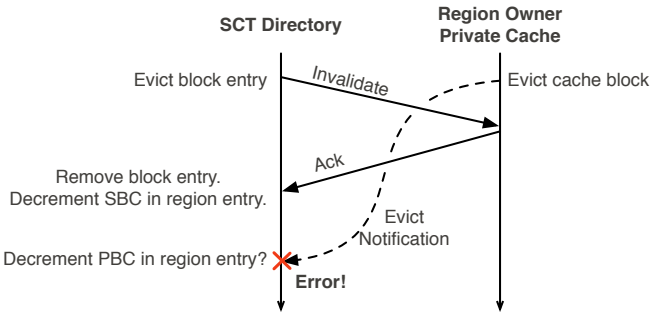


Figure 7: New protocol race with SCT.

SCT Structure: SCT describes neither the detailed structure it uses, nor the precise mechanism for accessing that structure. The paper implies the use of a simple set-associative structure for block and region entries. Many requests require multiple serial or parallel access to the SCT structure to find and update both block and region entries, thus increasing energy consumption and potentially latency. Also, as Section 5 shows, this organization, combined with the coupling block and region entries, results in high contention and many forced invalidations of cached blocks.

4.1 The DgD Approach

DgD avoids the difficulties of imprecision by using a precise present vector in region entries. Our initial exploration of the *i*DgD concept shows these vectors only need 16 to 64 bits, making them a viable and efficient alternative to counters. Precisely identifing every cached block means there is no need to scan the cache or directory, or to change the coherence protocol. As a result, DgD also avoids new race conditions.

In addition, DgD decouples block and region entries. Block entries can exist without any corresponding region entries, and in many cases block entries can be created or removed without modifiying the region entry.

Section 3 clearly describes the novel DgD structure which can efficiently locate both block and region entries with a single lookup by mapping regions and blocks to different ways for each region. Combining this with skewed-associativity and Zcache-style replacement allows DgD to make efficient use of all available directory capacity and minimize forced evictions.

DgD's unique structure, novel lookup mechanism, and precise region representation make it substantively different from SCT. As Section 5 shows, DgD provides better, more robust performance and increased energy efficiency.

5. EVALUATION

This section demonstrates DgD's effectiveness at reducing the directory size. Section 5.1 describes our methodology, and Section 5.2 describes the different configurations studied. Section 5.3 explores how small the DgD directory can be made without significantly increasing cache miss rates. Section 5.4 demonstrates that DgD performs better than a Sparse or SCT directory of the same size. Section 5.5

Table 1: Processor configuration

Processor Core	4 GHz UltraSPARC III ISA 8-stage, out-of-order 128-entry ROB, 64-entry LSQ decode/issue/commit any 4 instrs/cycle
Branch Predictor	8K GShare, 16K bi-modal, and 16K selector 2K entry, 16-way BTB, 2 branches/cycle
Fetch Unit	Up to 8 instrs/cycle, 32-entry fetch buffer
L1D/L1I	64 kB, 64 B blocks, 4-way, 2 cycle
Private L2	256 kB, 8-way, inclusive of L1 2/5-cycle tag/data latency
Shared L3	16 MB, 16-way, non-inclusive 3/9-cycle tag/data latency
Memory	16 GB, 4 channels, DDR3 1600 MHz

Table 2: Workload Descriptions

Online Transaction Processing (OLTP) — TPC-C	
db-a	64 clients, 100 warehouses (10GB)
db-b	16 clients, 100 warehouses (10GB)
Decision Support (DSS) — TPC-H	
qry2, qry6	commercial database system,
qry16, qry17	450 MB buffer pool
Web Server (Web) — SPECweb99	
apache	16K connections, FastCGI, worker threading
zeus	16K connections, FastCGI
Cloud — CloudSuite 1.0 [13]	
cassandra, classification, cloud9, nutch	
Mixed — Parsec 2.1 [6]	
dedup, fluidanimate, vips	simlarge input
blackscholes, canneal, swaptions	native input

details DgD's potential area and energy savings. Finally, Section 5.6 demonstrates the importance of using a single storage pool to store both block and region entries.

5.1 Methodology

We model a 16-core CMP using *Flexus* [28] and the full-system WindRiver Simics simulator [19]. The CMP uses a tiled layout where each tile has a processor core, private L1 and L2 caches, and a bank of the shared L3 cache. Tiles are connected by a mesh network with 128 bit links with a three cycle hop latency. The private L2 caches use a MESI coherence protocol, with the coherence directory co-located with the shared L3 banks. Additional parameters are described in Table 1. Table 2 lists the workloads studied. All simulated systems ran the Solaris operating system.

Sections 5.3 and 5.3.1 show results from trace-based simulations. Memory traces were collected while executing between 900 million and one billion instructions per core for each workload for a total of at least 14.4 billion instructions. Traces include both instruction and data references. The first five billion references in each trace were used to warm the cache hierarchies, and statistics were collected for the rest. The remaining sections show results for detailed timing simulations employing the SMARTS sampling methodology [30]. Functional simulations generated samples throughout program execution, storing cache and directory contents for each. The warmup for each sample ranges from 100 million to over a billion cycles depending on the workload and sample. Each sample measurement comprises 100k cycles of detailed warming followed by 100k cycles of measurement collection.

5.2 Directory Configurations

In addition to our proposed DgD design, we also modelled Sparse and SCT directories:

DgD: All designs have 4k sets and associativity between four and eight ways, and use 1 kB regions with sixteen 64 B blocks each. The resulting designs have between 0.25 and 0.5 entries for each block in the private caches. The largest design, $\text{DgD}_{8\text{way}}$, has half as many entries as a $\text{Sparse}_{1\times}$ design. We use a sixteen entry insertion queue. Sets are indexed using H_3 hash function based on irreducible polynomials [8].

Sparse: We use 8-way set-associative Sparse directories with full sharing vectors. A subscript indicates the degree of over-provisioning, e.g., $\text{Sparse}_{2\times}$ is two times over-provisioned, resulting in 16k sets.

SCT: We model an SCT directory [4] with the same size as $\text{DgD}_{8\text{way}}$. The original SCT paper omits many details, and we make a best effor attemtp to follow the implications of the paper without being prejudicial or disadvantaging SCT. We avoid adding optimizations not mentioned in the original paper. SCT uses 4K sets and 8-way set-associativity, the same as $\text{Sparse}_{0.5\times}$ and $\text{DgD}_{8\text{way}}$. Sets are indexed with low-order bits of the full block or region tag, similar to the Sparse designs. Requests first search for a block entry and then a region entry if necessary. This is more energy and area efficient than parallel accesses, and searching for a block first shortens the critical path for remote cache accesses. However, some workloads might prefer the opposite search order. New protocol races were handled by adding a structure at each directory tile to track races between evicts and invalidates. As in the original work, preference is given to evicting block instead of region entries. Unlike the original proposal, a single SCT directory tracks both instructions and data.

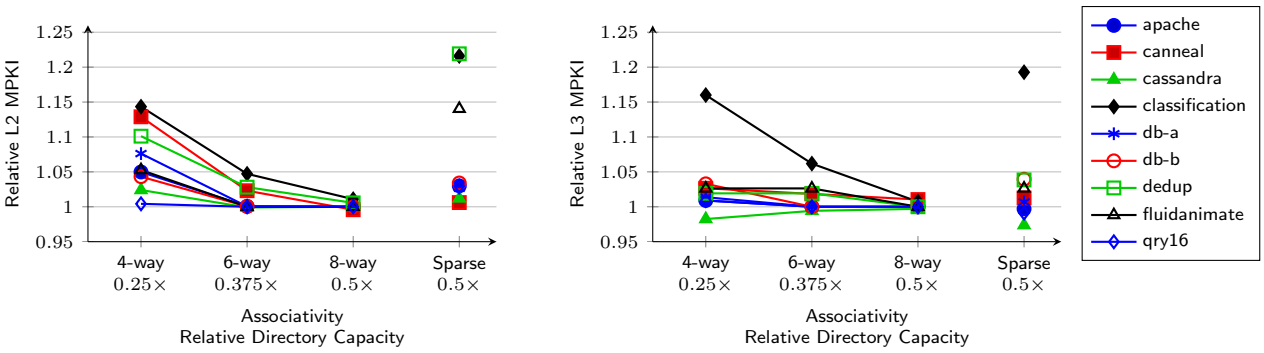
For all directory designs, we track coherence for both instructions and data using a single directory structure. A number of workloads experience writes to cache blocks that are also fetched as instructions, and our unified coherence protocol properly handles these events. We model a non-inclusive L3 cache, and blocks invalidated due to directory replacements are written to the L3 cache if not already present. All directory designs perform these write-backs.

5.3 DgD Size Exploration

To evaluate DgD size, we measure the private and shared cache miss-rates for different configurations. We vary the associativity between four and eight, and keep the number of sets fixed at 4K. Figure 8 shows representative results for a selection of workloads. The x-axis indicates the associativity and apacity of each design relative to a $\text{Sparse}_{1\times}$ directory. We measured the number of misses per thousand instructions (MPKI) for the private L2 caches (Figure 8a) and the shared L3 cache (Figure 8b), and the y-axis shows the MPKI relative to a Sparse_{∞} design with no conflict misses. As DgD associativity and size increase, miss rates decrease across all workloads, and an 8-way set-associative DgD barely affects the miss rate of the on-chip caches.

5.3.1 DgD vs. Sparse

To demonstrate the effects of under-provisioning a Sparse



(a) L2 MPKI

(b) L3 MPKI

Figure 8: Relative L2 and L3 MPKI for 4, 6, and 8-way DgD designs, and for a Sparse$_{0.5\times}$ directory, all with 4k sets.

directory, the right-most points in Figures 8a and 8b show the MPKIs for a Sparse$_{0.5\times}$ directory that has half as many entries as there are blocks in the private caches. This design has the same associativity and number of sets as the DgD$_{8way}$ design. The under-provisioning results in higher MPKIs – Sparse$_{0.5\times}$ increases the L2 MPKI for all workloads and by as much as 22% for classification and dedup. DgD$_{8way}$ only increases the L2 MPKI for two workloads, and by 1% or less.

Overall, DgD$_{6way}$ offers behavior comparable to Sparse$_{0.5\times}$ with 25% less storage. Meanwhile, DgD$_{8way}$ has a lower L2 MPKI than Sparse$_{0.5\times}$ for all workloads with the same storage requirements.

5.4 Performance

Figure 9a shows the performance of Sparse$_{0.5\times}$, SCT$_{8way}$, and DgD$_{8way}$ designs relative to Sparse$_{1\times}$. The harmonic mean speedup shows that DgD$_{8way}$ performs nearly identical to Sparse$_{1\times}$ on average, whereas Sparse$_{0.5\times}$ and SCT$_{8way}$ both suffer 2.7% slowdowns. Overall, these slowdowns appear lower than in prior works for several reasons: (1) Sparse$_{0.5\times}$ design is 8-way set-associative compared to 4-way for some prior works; (2) we model relatively large private L2 caches instead of small private L1 caches and shared L2 caches; (3) our directory tracks both data and instructions, and the many shared instructions reduce demand for directory resources; and (4) cache invalidations due to directory replacements cause write-backs to the shared L3 cache if the data is not already there. Rather than artificially inflate the effects of conflict misses, our design choices represent a realistic, high performance design.

In addition to having better average performance, DgD is also more robust. In the worst case, Sparse$_{0.5\times}$ and SCT$_{8way}$ both suffer slowdowns of over 16% for classification, while DgD$_{8way}$ has a speedup of 3.7% for this workload. The worst slowdown for DgD$_{8way}$ is only 4% for nutch and qry17. Figure 9a shows that a DgD design chosen to meet *average* storage requirements performs well across a variety of workloads and does not suffer from extremely poor performance for any workloads.

5.4.1 SCT vs. DgD Performance

Although SCT and DgD appear very similar on the surface, SCT consistently performs worse than DgD for a number of reasons:

Table 3: Directory area, energy, and power overheads.

	Area (mm^2)	**Energy** (pJ)	**Leakage** (mW)
Sparse$_{2\times}$	1.75×	1.55×	1.64×
Sparse$_{1\times}$	0.633	7.15	7.40
DgD$_{8way}$	0.59×	0.65×	0.67×
DgD$_{6way}$	0.40×	0.40×	0.51×

i) SCT sends speculative requests when the private block count is greater than zero. For canneal, 32% of requests to the L3 are forwarded to other private L2s, but 73% of these forwarded requests fail and must be satisfied from memory. This contributes to a 4% slowdown for canneal.

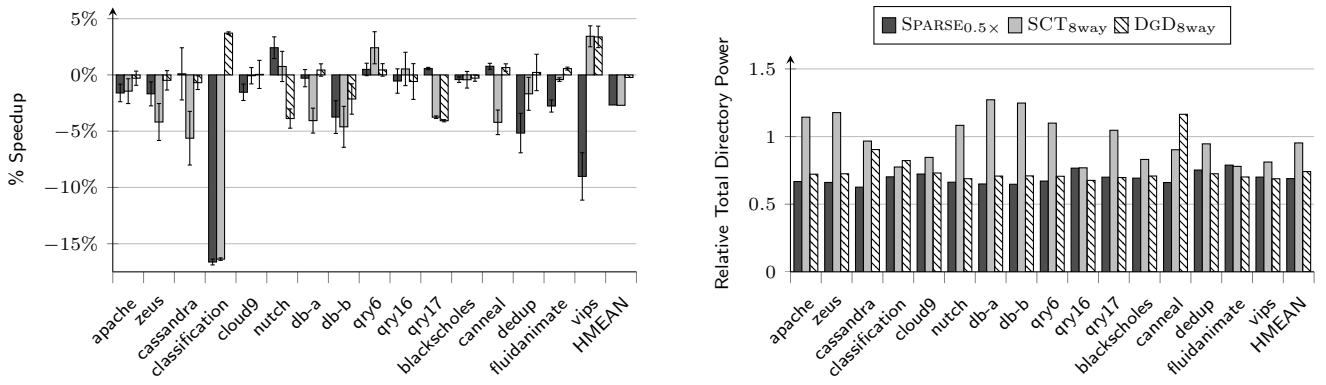
ii) SCT has more conflict misses. DgD uses more complex hash functions, skewed-associativity, and Zcache style replacement to reduce conflict misses. As Figure 10 shows, SCT occupies fewer directory entries than DgD on average, but conflict misses cause SCT to forcibly invalidate cache blocks roughly 3× more often than DgD. In addition, conflicts in SCT that evict a region entry also evict any block entries within that region, effectively causing *false* conflicts.

iii) Evicting region entries from SCT can require scanning the directory and one of the private caches, increasing contention for these resources. It also requires evicting all block entries in the same region, increasing the number of cache invalidations.

iv) SCT performs serial lookups to find block and region entries, and it often updates both block and region entries. This increases latency and directory contention.

5.5 Area & Energy

We used CACTI 6.5 to model various DgD and Sparse directories. Table 3 shows the estimated area, dynamic energy, and leakage power for various structures. All measurements are normalized with respect to Sparse$_{1\times}$. Each structure was optimized to reduce area while keeping the access latency and cycle time within three processor cycles. This allows the directory lookup to complete in parallel with the three cycle tag lookup for the shared cache.



(a) Performance

(b) Directory Power

Figure 9: Performance and directory power consumption relative to $\text{SPARSE}_{1\times}$ directory.

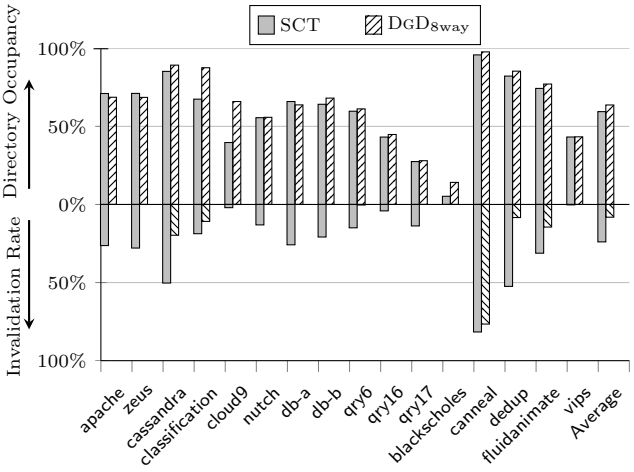


Figure 10: Upper columns show the percentage of directory entries used and lower columns show the percentage of directory allocations that evict cache blocks.

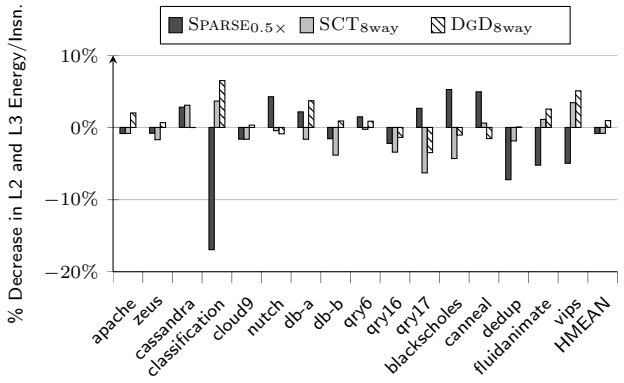


Figure 11: Percentage decrease in the total L2, L3 and directory energy per committed user instruction (excluding busy-waiting).

As Table 3 shows, $\text{DGD}_{8\text{way}}$ significantly reduces area, energy, and leakage power compared to $\text{SPARSE}_{1\times}$. Since DGD can perform many lookups when allocating and replacing entries, the energy used per access is not entirely representative of the total energy consumption. To account for such differences, Figure 9b shows the total directory power consumption relative to $\text{SPARSE}_{1\times}$ based upon the timing simulation results presented in Figure 9a. Overall, $\text{DGD}_{8\text{way}}$ reduces directory power by 25%.

$\text{SCT}_{8\text{way}}$ and $\text{DGD}_{8\text{way}}$ both have the same associativity, number of entries, and entry sizes; thus, the area, leakage power, and access energy estimates for $\text{DGD}_{8\text{way}}$ apply equally to $\text{SCT}_{8\text{way}}$. However, Figure 9b shows that $\text{SCT}_{8\text{way}}$ actually *increases* directory power by over 20% for some workloads where a majority of requests need multiple lookups to find and modify both the block and region entries.

The performance loss, forced invalidations, and speculative snoops of $\text{SCT}_{8\text{way}}$ also affect the energy consumed by the various on-chip caches. We measure these effects and calculate the total amount of energy consumed by the directory, the L2 caches, and the L3 caches, and divide this amount by the number of committed user instructions. Figure 11 shows the resulting decreasing in energy per instruction for each design (higher positive numbers are better). As the figure shows, $\text{SCT}_{8\text{way}}$ increases the L2, L3 and directory energy per instruction by 0.8% on average and up to 6% in the worst case, while $\text{DGD}_{8\text{way}}$ *decreases* the average energy per instruction by 1% and never increases it by more than 3.5%. Thus, $\text{DGD}_{8\text{way}}$ is measurably more energy efficient on average.

5.6 Importance of Common Storage

DGD uses one structure for both block and region entries, and it imposes no restriction on how many block or region entries exist at a given time. Figure 12 demonstrates this feature's importance by showing a breakdown of the average contents of $\text{DGD}_{8\text{way}}$ for each workload. Some workloads, such as dedup, mostly store block entries, while others, such as classification, store mostly region entries. Using separate structures for block and region entries would require over 50% more regions to accommodate these extremes.

Figure 12 also indicates the degree of spatial locality experienced by temporarily private regions. Region entries dominate for both blackscholes and classification, but while

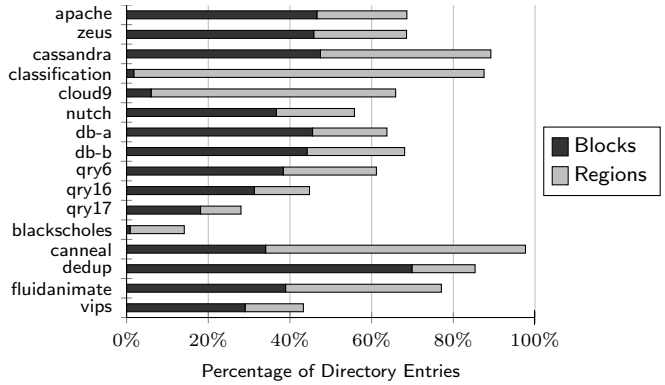


Figure 12: Breakdown of the average number of block and region entries within the DGD_{8way} directory.

blackscholes has significant spatial locality and uses a small portion of the directory, classification has much less spatial locality and requires many more region entries. This, even with low spatial locality, DGD significantly reduces the directory size without reducing performance.

5.7 DgD vs. an Improved SCT

The SCT_{8way} design evaluated above attempts to closely match the original design proposed by Alisafee [4]. We also evaluated a more optimized SCT-Z_{8way} design that borrows many of the implementation details of DGD_{8way}. This new design uses DGD's way-mapping technique to lookup block and region entries in parallel, along with a Zcache-style replacement policy and a 1 kB region size. With these improvements, SCT-Z_{8way} is able to make more effective use of the available directory capacity, resulting in a higher average occupancy and fewer cache block invalidations than SCT_{8way}. This improves the average performance, resulting in just a 0.6% slowdown compared to the average 2.7% slowdown experienced by the original SCT_{8way} design. Despite these improvements, SCT-Z_{8way} still suffers from the need to store a region entry for every block. As a result, classification suffers from a 5.7% slowdown, and canneal increases the combined L2, L3 and directory energy per instruction by 12%. Thus, while Zcache-style replacement and other optimizations help improve overall performance, they are not sufficient to provide the same robustness as the DGD design.

6. RELATED WORK

Existing work has explored many ways to reduce the size of coherence directories.

Several works have focused on reducing the size of directory entries using formats such as coarse vectors [16], segment directories [9], the SGI Origin directory [18], and limited pointer directories [3]. More recently, SPACE [33] stores sharing patterns in a separate table and each entry stores a small pointer into this table. These schemes only accurately represent a limited number of possible sharing patterns, and the directory either restricts sharing or uses imprecise information causing unnecessary network traffic.

Ferdman *et al.* propose using Cuckoo hashing to reduce conflict misses in sparse directories [12]. This reduces overprovisioning, but the directory still requires 1× to 1.5× as many entries as private cache blocks, while DGD only needs 0.5× as many entries.

Scalable Coherence Directories (SCD) [24] scale well up to a thousand cores or more. *Root* entries contain pointers to a few potential sharers, and blocks with more sharers use a hierarchy of multiple directory entries. SCD uses skewed-associativity and Zcache-style replacement, similar to DGD. DGD is orthogonal to SCD: where SCD reduces the space required for large sharing vectors, DGD reduces the required number of directory entries. As Section 3.4.5 discusses, DGD and SCD can be combined to further improve scalability.

Tagless coherence directories [32] use bloom filters to track which blocks are in the private caches. This approach significantly reduces directory storage requirements, but the access energy does not scale well to large core counts. Zhao *et al.* propose combining Tagless and SPACE directories [34]. The resulting design improves energy efficiency at the cost of decreased precision in the representation of sharing patterns.

Cuesta *et al.* propose deactivating coherence for private pages of memory and achieve similar benefits to DGD by reducing the number of directory entries [11]. However, this scheme uses TLBs to track shared and private pages, and so is not transparent to software. It also requires scanning the cache when a page transitions from private to shared, and either flushing blocks or recording them in the directory. Hardavellas *et al.*, propose Reactive NUCA (R-NUCA) which uses a similar scheme to optimize the placement of data in a distributed shared cache [17]. In contrast to both schemes, DGD is transparent to software, including the OS, and reduces the overhead of tracking regions that are only temporarily private, even if they are shared over a longer period of time. However, since private regions occupy entries in the DGD directory, the other schemes might provide more benefits for some workloads.

Ros and Kaxiras propose a directory-less coherence protocol that uses write-through for shared pages and write-back for private pages [22]. The VIPS-M protocol also flushes shared data from private caches at synchronization points. It requires scanning and flushing private caches when a page transitions from private to shared. Since pages accessed by more than one core are permanently marked shared, some workloads where the majority of pages are shared might suffer using this approach. Also, workloads with frequent, fine-grain synchronization might cause excessive scanning and flushing of caches, possibly reducing performance.

RegionScout [21] and Coarse-Grain Coherence Tracking [7] avoid broadcasts in a snoop coherence protocol for requests to temporarily private regions. DGD exploits this intuition to reduce the storage requirements for directory coherence.

7. CONCLUSION

Multi-grain coherence directories offer an innovative mechanism to reduce directory size beyond the 1:1 ratio traditionally required of SPARSE directories. MGDs allow each entry to track a large, temporarily private memory region, instead of just tracking one cache block, thus reducing the required number of directory entries. Our investigation of an ideal MGD that dynamically refines the coherence granularity reveals that a dual-grain directory that tracks individual cache blocks and coarse-grain regions of 1 kB to 4 kB offers the greatest potential benefit for a variety of workloads.

DGD is a practical implementation that reduces the directory area by 41% to 66% compared to sparse directories that are 1× and 2× over-provisioned. DGD requires no coherence protocol modifications, introduces no unnecessary snoop bandwidth, adds no extra latency to requests, and is entirely transparent to software. Compared to the previously proposed SCT directory, DGD performs better and uses less energy. Future work may investigate combining DGD and SCD and consider whether DGD's region-level information can enable other optimizations.

8. REFERENCES

[1] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). White Paper, 2008.

[2] OpenSPARC^{TM} T2 system-on-chip (SoC) microarchitecture specification, May 2008.

[3] A. Agarwal et al. An evaluation of directory schemes for cache coherence. In *Proc. of the Int'l Symposium on Computer Architecture*, June 1988.

[4] M. Alisafaee. Spatiotemporal coherence tracking. In *Proc of the Int'l Symposium on Microarchitecture*, Dec. 2012.

[5] L. A. Barroso et al. Piranha: a scalable architecture base on single-chip multiprocessing. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2005.

[6] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2005.

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

[9] J. H. Choi and K. H. Park. Segment directory enhancing the limited directory cache coherence schemes. In *Proc. of the Int'l Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 258–267, Apr 1999.

[10] G. Chrysos. Intel® many integrated core architecture: The first Intel® Xeon Phi coprocessor (codenamed Knights Corner). presented at Hot Chips 24, Stanford, CA, Aug. 2012.

[11] B. A. Cuesta et al. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proc. of the Int'l Symposium on Computer Architecture*, 2011.

[12] M. Ferdman et al. Cuckoo directory: A scalable directory for many-core systems. In *Proc. of the Int'l Symposium on High Performance Computer Architecture*, Feb. 2011.

[13] M. Ferdman et al. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[14] G. Grohoski. Niagara2: A highly-threaded server-on-a-chip. presented at Hot Chips 18, Stanford, CA, Aug. 2006.

[15] S.-L. Guo et al. Hierarchical cache directory for CMP. *Journal of Computer Science and Technology*, 25:246–256, 2010.

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

[17] N. Hardavellas et al. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proc. of the Int'l Symposium on Computer Architecture*, 2009.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proc. of the Int'l Symposium on Computer Architecture*, June 1997.

[19] P. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[20] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[21] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2005.

[22] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *Proc of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2012.

[23] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proc. of the Int'l Symp. on Microarchitecture*, Dec. 2010.

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

[25] A. Seznec. A case for two-way skewed-associative caches. In *Proc. of the Int'l Symposium on Computer Architecture*, 1993.

[26] S. Turullols and R. Sivaramakrishnan. SPARC T5: 16-core CMT processor with glueless 1-hop scaling to 8-sockets. presented at Hot Chips 24, Stanford, CA, Aug. 2012.

[27] D. A. Wallach. PHD: A hierarchical cache coherent protocol. Technical report, Cambridge, MA, USA, 1992.

[28] T. F. Wenisch et al. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[29] B. Wheeler. Tilera sees opening in clouds. *Microprocessor Report*, 25(7):13–16, July 2011.

[30] R. E. Wunderlich et al. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. of the Int'l Symposium on Computer Architecture*, June 2003.

[31] Q. Yang, G. Thangadurai, and L. M. Bhuyan. Design of an adaptive cache coherence protocol for large scale multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 3(3):281–293, May 1992.

[32] J. Zebchuk et al. A tagless coherence directory. In *Proc. of the Int'l Symposium on Microarchitecture*, Dec. 2009.

[33] H. Zhao et al. SPACE: sharing pattern-based directory coherence for multicore scalability. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2010.

[34] H. Zhao et al. Spatl: Honey, i shrunk the coherence directory. In *Proc of the 2011 Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2011.