Analysis and Implementation Alternatives of Multi-grain Coherence Directories

Víctor Galindo-Garre, Rubén Titos-Gil, Ricardo Fernández-Pascual, and Alberto Ros

***Abstract*—Modern chip multiprocessors continue to scale in core count and private cache capacity, increasing storage pressure on coherence directories. Prior work has addressed this challenge through compressed sharing information, sparse directories, and coarse-grain tracking. Multi-grain directories reduce storage requirements while preserving tracking precision by supporting dual granularity: block and region. At region granularity, multiple private blocks held by the same core are represented by a single entry. Yet, existing designs rely on complex structures that hinder practical implementation, and their trade-offs remain insufficiently characterized.**

This work provides a detailed characterization of multi-grain coherence directories. Unlike conventional single-grain designs, these directories impose constraints on how blocks map to distributed shared cache banks. We analyze alternative mappings and their implications for directory organization. Based on this analysis, we introduce minimal modifications that enable entry merging comparable to state-of-the-art multi-grain designs without requiring complex structures. Our results show that this simplified design matches state-of-the-art performance while significantly reducing implementation complexity.

1. Introduction

The sustained growth in core counts and per-core private cache capacity of server-class chip multiprocessors makes maintaining cache coherence progressively more costly [1]. Directory-based protocols [2, 3] tackle this by maintaining a centralized record of which cores hold each cached block. However, the storage that this record demands grows with aggregate private capacity. Contemporary server chips already integrate 128–192 cores [4, 5, 6], each backed by several megabytes of private cache, requiring the directory to track millions of 64-byte blocks. Decoupling the directory from the last level cache (LLC) through a sparse design [7, 8] keeps the entry count low, yet this count stays proportional to the aggregate capacity of the private caches; a directory provisioned for worst-case occupancy thus dedicates a substantial fraction of on-die area to entries that remain idle throughout most of the execution.

Multi-grain directories [9, 10, 11, 12] approach this problem from a different angle: rather than always allocating one entry per cached block, they also allow grouping all private blocks with the same owner within a contiguous address range into a single *region* entry—a shorter tag, an owner pointer, and a per-block presence bitmap. On workloads dominated by private data, this coarser granularity allows grouping many block entries into a single region entry, reducing the number of directory entries needed to keep track of each coherent block. In this way, they reduce storage while preserving tracking precision.

The state-of-the-art implementation of this approach is the Dual-Grain Directory (DGD) [9], a multi-grain directory that fixes the region size to the default memory page used by the operating system. This directory design achieves high region-entry utilization through a combination of three mechanisms: a constrained address-to-home mapping that forces all blocks belonging to the same region to share the same home directory bank, a per-way hash function that partitions block and region entries throughout the associativity, and a ZCache [13] that evicts and relocates conflicting entries. The latter mechanisms add non-trivial implementation complexity [14, 15, 16], yet DGD has been evaluated only as an integrated system, leaving it unclear which mechanisms drive the observed gains and whether the same benefit could be obtained with a simpler design.

This paper argues that by extracting the core principles of DGD, it is possible to deliver comparable performance through a much simpler implementation. To do so, this article isolates the contribution of every mechanism in DGD (and of the address mappings that region entries silently demand) to design *CE-DGD*, a cost-effective dual-grain directory built atop a conventional set-associative cache. On a 128-core processor running SPLASH-4 and PARSEC benchmarks, CE-DGD captures essentially all of DGD's performance while requiring less area and complexity.

This paper makes the following contributions.

- We characterize the impact of the address mappings required by region-based directories and propose a scheme that maps contiguous blocks to the same directory set. We show that, for small regions, this mapping achieves performance comparable to the standard multi-grain mapping. (Section 3)
- We decompose DGD into its constituent mechanisms, i.e., way distribution, ZCache, dual-access lookup, and static partitioning, and measure the performance impact of each contribution. (Section 4)
- We propose CE-DGD, a cost-effective dual-grain directory built on a conventional set-associative organization, which is enabled by the directory mapping proposed in this work and reduces the complexity

• *V. Galindo-Garre, R. Titos-Gil, R. Fernández-Pascual and A. Ros are with the Departamento de Ingeniería y Tecnología de Computadores, University of Murcia, Spain.*
E-mail: victor.galindog@um.es, rtitos@um.es, rfernandez@ditec.um.es, aros@ditec.um.es

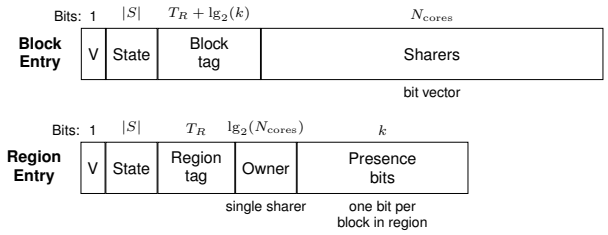


Figure 1. Entry formats of a DGD directory entry

of the original DGD proposal. (Section 5)

- We discuss how CE-DGD can be adapted to the more complex address-hashing schemes used by modern processors. (Section 6)
- We evaluate the design on a 128-core processor running SPLASH-4 and PARSEC benchmarks, including a tag-storage analysis. (Sections 7 and 8)

2. Background

2.1. Sparse vs. tag-embedded directories

Inclusive last-level caches (LLCs) naturally support a *tag-embedded* directory, where each LLC entry stores its coherence state and sharer vector [17]. Although this approach amortizes tag storage, it is inefficient because LLCs contain many more lines than private caches, and most entries do not carry coherence information. Sparse directories [7, 8] decouple the directory from the LLC and size it according to the aggregate private cache, enabling support for non-inclusive LLCs. As a result, they are the de-facto choice for many-core systems and serve as the baseline in this work. However, sparse directories still require one entry per private cache block, which becomes prohibitive as core counts and private cache sizes scale.

2.2. Region-based coherence directories

The DGD directory [9] uses two types of directory entry: block and region (see Fig. 1). *Block entries* track the sharers of a single block with a bit vector. *Region entries* track a group of private blocks within an address range with a pointer of $\log_2(N_{\text{cores}})$ bits. They also include a presence bitmap that indicates which blocks of the region are tracked by the entry. Region entries use a shorter tag than block entries, and the size of the region matches the granularity of the OS page, so that block and region entries have comparable sizes despite their different formats.

It is important to remark that the region size does not need to match the OS page size, but as shown in [9], matching the OS page size helps to improve performance on workloads where private data structures prevail (e.g., partial results stored in a per-thread matrix or the process stack).

To support both formats in a single access, DGD relies on a ZCache [13] with a constrained per-way hash function. ZCache look-ups use a different hash per way and resolve insertion conflicts through breadth-first relocation of resident entries, reducing evictions at the cost of requiring breadth-first search (BFS) traversal hardware and an adapted replacement policy.

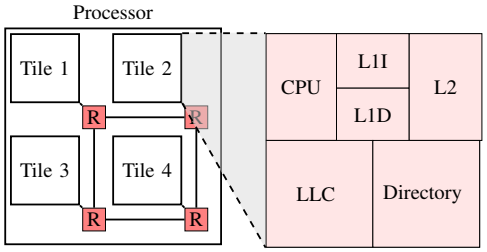


Figure 2. Multicore organization considered in this work

Apart from the directory organization, region-based directories impose a key requirement: To avoid the substantial overhead of cross-bank coordination when creating region entries, region-based directories strictly require all blocks within a given region to be mapped to (homed at) the same directory bank. This constraint fundamentally alters the block mapping policy compared to a conventional block-interleaved baseline, serving as the foundational premise for our subsequent analysis.

3. Analysis of Mappings to LLC and Directory

This work considers a conventional tiled multicore memory hierarchy, as illustrated in Fig. 2. Each tile comprises private L1 and L2 caches. The shared LLC is partitioned into banks distributed across tiles. Coherence among private caches is maintained using a sparse distributed directory co-located with the LLC banks. Each directory entry includes a valid bit, tag, state, and sharer vector. The tiles are connected through a 2D-mesh network.

In a shared distributed LLC and directory design, each memory block is assigned to a home bank, where its data and coherence information reside. Traditional directory designs use a static home mapping to determine this assignment. This choice affects performance in two ways: it determines access latency by selecting where requests are served, and it influences conflict behavior by distributing blocks across sets in LLC and directory banks. Poor mappings can increase directory pressure, leading to directory-induced invalidations that evict actively used private-cache blocks.

To maximize efficiency, the LLC and the directory typically share the same home mapping. This alignment enables parallel access to both structures, allowing coherence checks and data retrieval in a single step. In contrast, mismatched mappings require additional hops to access the data and update the directory, increasing both latency and complexity.

In conventional set-associative caches, each address is assigned to a single set. Although alternative designs allow for more flexible placement across sets [13, 14, 15], our proposal assumes a conventional organization and evaluates its impact on directory performance.

Mapping addresses to distributed LLC banks and directory slices becomes particularly important in region-based directories. These designs impose stricter constraints than block-granularity approaches: all blocks within a region must map to the same directory home to preserve correctness. In our design, this constraint is further strengthened,

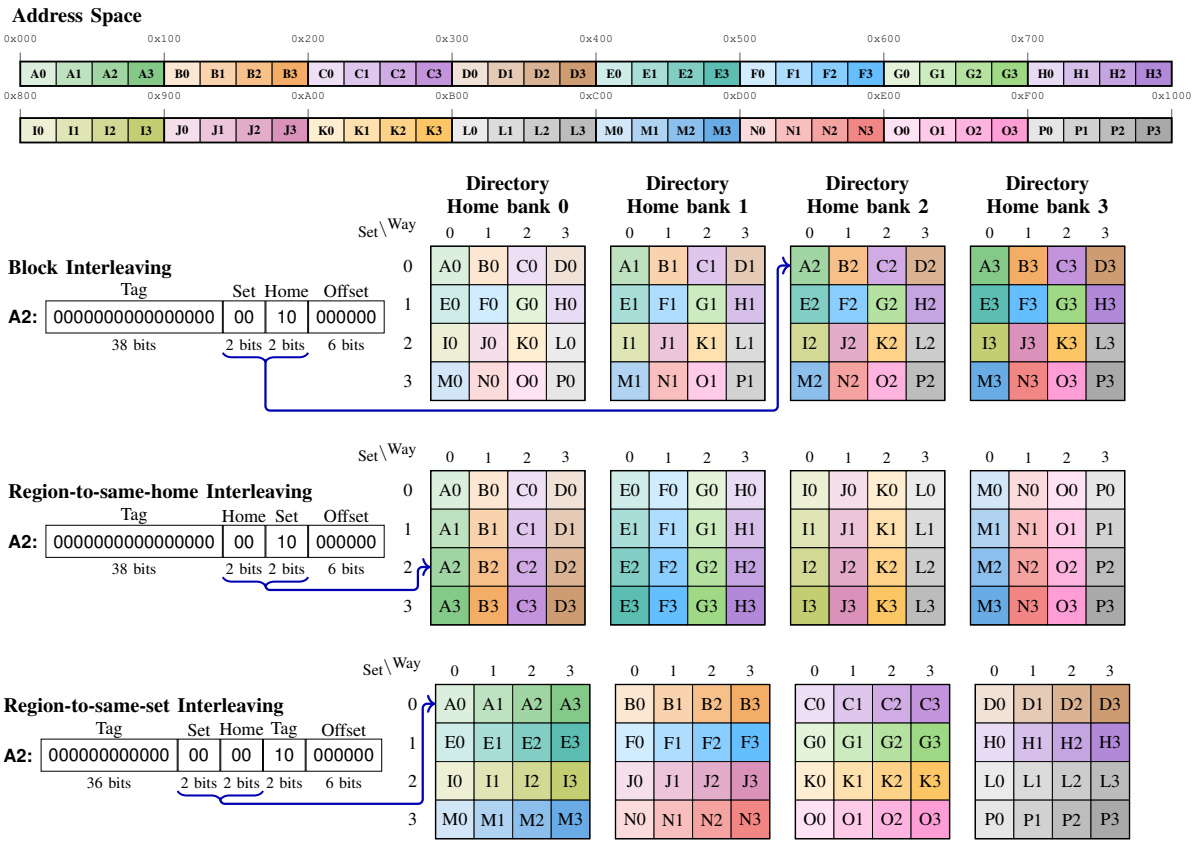


Figure 3. Mapping on a distributed directory under (a) the block-interleaved baseline, (b) region-to-same-home interleaving, and (c) region-to-same-set interleaving

requiring that all blocks within a region also map to the same directory set, which simplifies the directory organization compared to prior work.

Changing the address mapping can impact the performance of the system even without modifying the directory design. To isolate this effect, we evaluate how different mappings influence the execution time of the benchmarks. Specifically, we consider three mapping strategies, illustrated in Fig. 3: block interleaving, region-to-same-home interleaving, and region-to-same-set interleaving.

3.1. Baseline mapping: Block interleaving

The baseline mapping used in this work is *block interleaving* (first mapping in Fig. 3), which assigns contiguous blocks to successive LLC banks. This mapping is applied to both the LLC and the directory; however, we focus on the directory mapping for clarity. The lower $\log_2(block\ size)$ bits of the physical address define the block offset. The next $\log_2(cores)$ bits select the home bank, followed by the $\log_2(sets\ of\ dir)$ bits that index the set, with the remaining higher-order bits forming the tag. The mapping for the physical address 0x08C is illustrated in the figure, and explained below (from least to most significant):

- Block offset: considering a standard 64-byte block system, the $\log_2(64) = 6$ lower bits map each byte of the block.
- Home: in the 4-tile system of the example, the next $\log_2(4) = 2$ bits select the bank where this block is stored (for address 0x08C, bank 2).
- Set: in the 4-set directory of the example, the next $\log_2(4) = 2$ bits select the set inside the home directory (for address 0x08C, set 0).
- Tag: the remaining bits are used as tag to identify the block.

We can also observe in the example that adjacent blocks $A_0 - A_3$, corresponding to physical addresses {*0x000, 0x040, 0x080, 0x0C0*}, are assigned to different banks. As mentioned previously, it is important to note that the LLC follows essentially the same mapping as the directory. However, since the number of sets in the LLC may differ from that of the directory, the number of set bits can change, but their starting position in the address remains the same. If a non-bijective hashing scheme is used, as in skewed caches [15], the set bits are combined with way-specific hash functions rather than mapped directly to a single set.

3.2. Region-to-same-home mapping

Region-based directories require the mapping to directory banks to be adapted so that all blocks within a region are mapped to the same bank. To achieve this, the home

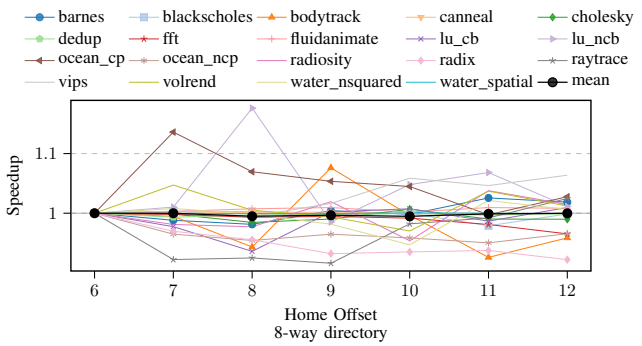


Figure 4. Speedup as a function of the home offset for an 8-way directory, normalized to a block interleaving baseline

bits are moved toward the most significant bits, depending on how many adjacent blocks are placed in the same bank.

This region-to-same-home mapping (or in short same-home mapping) uses the bits immediately following the block offset to index the cache set, rather than the home node. The number of set bits depends on how many adjacent blocks are placed in the same bank. For example, if a region consists of four memory blocks, i.e., a 256-byte region, the home bits start at bit 8, which is two bits ($\log_2(4) = 2$) after the 64-byte block offset. In many cases, the number of bits used for the home is smaller than the number of bits used for the set. In such cases, the set bits continue after the home bits, and the remaining higher-order bits form the tag (the set bits are split in two parts).

Fig. 3 depicts the case in which 256-byte regions $A_0 - A_3$ are placed inside the same directory bank, but in different sets. Below the set and home selection is described:

- Home: should be shifted as many bits as the $\log_2(region\ size)$. In a 256-byte region, it is shifted 8 bits; in a 1 KB region, it is shifted 10 bits. In our example, address 0x08C maps now to bank 0.
- Set: the least significant bits of the block address (from the block offset on) that do not identify the home are used to determine the set inside a bank. In our example, address 0x08C maps now to set 2.

As in the block-interleaved mapping, the same bits must be used to select the home bank in both the directory and the LLC. Otherwise, communication traffic and execution time of applications would increase significantly.

Changing the mapping of blocks to LLC banks can affect the performance of the system [18]. For this reason, we evaluate the impact of changing the home-offset from a 6-bit shift (i.e., baseline block interleaving) to a 12-bit shift (where 4 KB regions are mapped to the same home). The results of this analysis, shown in Fig. 4, are performed on an 8-way directory where the directory has the same entries as the private L2 and the rest of the system is the same as described in Section 7. The speedup is nearly identical for 16- and 32-way directories.

On average, modifying the home bits using different home offsets has a small impact on execution time. However, two benchmarks show noticeable variations in opposite directions: `lu_cb` and `ocean_cp`. In `lu_cb`, as the home offset increases, LLC latency increases, resulting in higher execution times. In `ocean_cp`, the time spent on synchronization decreases as the offset increases, meaning that some threads finish earlier than others. These variations are not due to one mapping being inherently better or worse, but rather to the relative proximity of home banks to the cores issuing requests that miss in the private caches. Since this proximity is essentially random, some applications can benefit from a lucky placement and run faster, while others may suffer from a worse placement. Average, however, shows no important variations.

3.3. Region-to-same-set mapping

The same-home mapping is needed for multi-grain directory approaches. However, as mentioned, these proposals are based on complex directory cache designs. If a simple cache structure was used for the directory, a multi-grain directory approach would be inefficient, as block entries would map to different sets than region entries. We elaborate on this problem in Section 4.

Hence, with the goal of having block and region entries not only in the same home, but also in the same directory set, as required by our proposal (Section 5), we present a different mapping named region-to-same-set (or in short same-set). As in the same-home mapping, the home bits do not change position. However, the bits used to select the directory set are placed after the home bits. This allows contiguous entries to reside within the same directory set (of the same bank). The bits between the home and the block offset are incorporated into the tag.

Again, the example is shown in Fig. 3: the first 6 bits form the block offset, the home bits start at bit 8 (allowing 256-byte regions to reside in the same bank), and the set-index bits follow the home bits (forcing the 256-byte regions to the same set). The example illustrates how adjacent addresses $[A_0 - A_3]$ are now placed within the same directory set.

However, this mapping can be critical for performance, if no restrictions are given. We evaluate this mapping for 8-, 16-, and 32-way directories, again varying the size of the regions that are mapped to the same home and set (from 6 bits –64 bytes– to 12 bits –4 KB– regions, x axis). Fig. 5 shows that for small regions the average performance does not vary significantly (again, with some outliers attributed to random mapping effects). However, performance decreases significantly for large region sizes. For the 8-way directory, where each set can track 8 blocks (a region of 512 bytes), using this mapping for regions greater than 512 bytes degrades its performance significantly. A similar trend happens for the 16-way directory for regions larger than 16 blocks, i.e., 1 KB, and for the 32-way directory for regions larger than 32 blocks, i.e., 2 KB.

The performance drop occurs because larger regions increase the probability of conflicts within the same set. For example, in a 4-way directory with regions larger than 256 bytes, e.g., 512 bytes (8 consecutive blocks mapped to the same set), after the first 4 blocks are placed in a set, the next

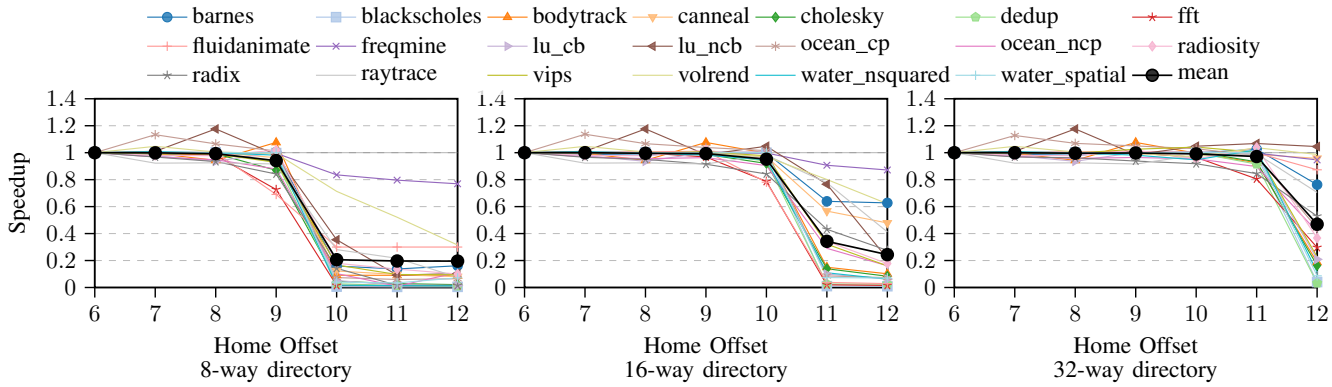


Figure 5. Performance normalized to block interleaving for the same-set mapping for 8-, 16-, and 32-way directories as the region size grows

4 blocks would evict them, increasing directory-induced invalidations and undermining spatial locality across threads. That is, conflict misses in the directory increase.

When the region size equals the region tracked by the directory set, performance can still be degraded because conflicting blocks are likely to evict each other. In this case, all the blocks in a region can be tracked by the directory, but if two regions map to the same set (e.g., from different threads and having private blocks), then directory conflicts explode. This is visible in `fft` with the 8-way directory: when the directory set size equals the region size (home offset = 9), speedup drops by more than 20%.

4. Analysis of the Dual-Grain Directory (DGD)

Once we have analyzed how different mappings affect region-based directories, we break down each of the techniques described in DGD, and propose a simpler design. As previously shown in Fig. 1, the DGD directory uses two entry formats: block entries and region entries of 4 KB.

4.1. Distribution of block and region entries

The DGD directory performs a single lookup to find both the block and the region entries assigned to the target address. This cannot be achieved with a conventional cache (unless a same-set mapping is employed); instead, it requires the modified skewed-access design of ZCache, which distributes region and block entries across different ways.

On a lookup, given the target address, DGD selects the ways where either a block or a region for that address can be placed. If the selected way is for a block entry, the block address (removing the home bits) is hashed to determine the set within that way. If the selected entry is for a region, the region address (removing the home bits) is hashed instead. This design intentionally uses different hash functions for region and block entries to prevent blocks within the same region from conflicting. For example, if different blocks within a region cannot coexist in the same region entry (e.g., because they are shared), each block can still be placed in the same way (but of another set), since the hash function uses the block address rather than the region address.

However, distributing a region entry and a block entry across each way is non-trivial. For a w-way directory, the hash function must be able to select half of the ways to find a block entry and the other half to find a region entry. This function should select one of the $\binom{w}{w/2}$ possible partitions. As there is no specification on how it is designed in the original article, we first assume an ideal distribution: when we want to find an entry, we pseudo-randomly select one of the available partitions. Even if it is pseudo-random, this distribution is deterministic, meaning that given a region address, it always selects the same partition.

In practice, a function able to distribute regions and blocks equally between ways would increase significantly the complexity of the hardware implementation. We propose a simpler design that is easier to implement to study its impact on performance.

We evaluate a function that distributes blocks and region entries across ways using two partitions defined on Eq. 1. This DIST function assigns a subset of ways to search a block depending on the result of hashing the region tag ($H_3(reg_tag)$). If $H_3(region_tag)$ returns zero, the first half of the ways are used to search for a block entry, and the remaining half for a region entry during a lookup. If $H_3(region_tag)$ returns one, $\{0,1,2,4\}$ ways are used to find a block and the other ways to find a region entry. This design is more flexible than the SS-DGD [10] that uses two caches, one for blocks and another for regions.

$$\text{DIST}(\text{reg_tag}) = \begin{cases} \{0,1,2,3\}, & \text{if } H_3(\text{reg_tag}) = 0 \\ \{0,1,2,4\}, & \text{if } H_3(\text{reg_tag}) = 1 \end{cases} \quad (1)$$

4.2. ZCache vs. Skewed cache

DGD relies on a ZCache-based lookup, which uses a different hash function per way. In addition, on insertion, it can displace entries to make room for new ones using a breadth-first search (BFS) to swap entries.

The ZCache reduces evictions when the directory occupancy is high, a regime typical of private-block-dominated workloads; as sharing grows, fewer entries are needed and the benefit of relocation decreases. This is because a ZCache behaves similarly to a highly associative cache, due to its ability to relocate existing entries when it needs to allocate a new one in a set that is full. However, with high directory

pressure, there is a certain point at which searching for an empty entry is not worth the effort, either because no candidate is found to replace the entry, or because the per-set hash function already provides a good distribution, making relocation unnecessary. For this reason, we evaluate in Section 8 whether the benefit comes in DGD from the skewed access to cache alone (e.g., a Skewed cache) or from the replacement policy of the ZCache, by removing the BFS traversal but keeping the per-way hash functions.

4.3. Dual-access directory

A skewed cache is still more complex than a traditional cache. DGD can be implemented on top of a traditional cache. However, in this design, when a same-home mapping is employed, a block entry could reside in a different set than its corresponding region entry. This design, while simpler than the original one, would require two ports to access both sets in parallel, doubling the directory cost. An alternative would be to use way-partitioned caches [19, 20, 21], but this would limit the number of block or region entries that can coexist in the directory. Another drawback is that it cannot support *lazy merging*, by which a block entry sharing an owner with a resident region entry is folded into it on eviction. Lazy merging captures regions whose blocks are not installed simultaneously, and it could be essential to reduce directory-induced invalidations.

5. A Cost-Effective Dual-Grain Directory

The DGD design has three properties: (1) it supports 4K-sized regions to minimize the number of entries, (2) it uniformly distributes regions and blocks across directory ways, and (3) it minimizes directory evictions. While it relies on a ZCache to support 4K-sized regions, the need to reallocate blocks to reduce conflicts and distribute blocks and regions across each way significantly increases design complexity.

For this reason, we propose a DGD variant that uses a conventional cache, which we call Cost-Effective Dual-Grain Directory (CE-DGD). As the directory associativity increases, a ZCache is no longer needed to avoid replacements. In addition, if a same-set mapping is employed, the entries are naturally distributed as needed within the same set. On the other hand, relying on associativity may require smaller region sizes, but, as we show in this work, large regions are not required to achieve low directory utilization.

5.1. Design principles

The Cost-Effective Dual-Grain Directory (CE-DGD) is built on two design choices: (1) all block and region entries of the same region are mapped to the same directory set and the same home bank (same-set mapping); (2) the region size in blocks is smaller than or equal to the directory associativity. The first choice enables a single-set lookup for both formats and removes both the per-way hash and the ZCache. The second choice is the key design parameter: with w ways and a region of r blocks, up to w/r regions can coexist in a set. A larger r raises the compression factor of the directory; a smaller r trades compression for robustness. For example, with regions up to 4 KB, in the best scenario, we could compress 64 blocks of 64 bytes into a single region entry. However, in the worst case where none of them can be compressed, it is necessary to store each of the blocks in the same set as seen in Section 3.3. If this worst case cannot be avoided, then blocks within the same region could be evicted on contiguous accessing as they cannot be placed in the directory at the same time. This is one of the limitations encountered, however, as the associativity increases, the size of each region can increase without encountering this pathology.

5.2. Operation

A CE-DGD lookup examines a single set using a same-set mapping, checking all w ways in parallel. Instead of distributing half of the ways with block formats and half with region formats, as the same mapping is used for both, entries of the same region can be mapped to the same ways whether they are block or region entries. The per-way format bit selects whether the entry is compared as a block or as a region. In a region, checking the presence bits allows checking whether the block is present inside the region.

Two operations require special handling, as in the DGD. When a coherence event adds a second sharer to a block represented by a region entry, CE-DGD demotes the block to a dedicated block entry and clears the presence bit. On the eviction of a block entry, CE-DGD opportunistically folds it into a region entry of the same set sharing its owner; because the lookup already inspected the entire set, this merge introduces no additional access. Otherwise, the entry is evicted using the directory's normal replacement policy (e.g., LRU).

5.3. Storage analysis

Because CE-DGD uses a simple set indexing, the set-index bits are removed from the tag as in a traditional cache. However, in a per-way hashed cache those bits should be maintained. For a 48-bit physical address space, 64 B blocks, 128 tiles, and a 16-way directory, the tag shrinks to $\sim$24 bits per entry, against 35 bits in the original DGD design (whose ZCache prevents this reuse). On the evaluated 256 KB directories the saving is only 4.85 %, but on future chips with 2 MB private caches it would reduce the per-bank tag footprint from 660 KB to 616 KB.

Table 1 reports the tag width and footprint as a function of the directory size for a 16-way directory with 128 tiles, 64-byte blocks, and a 48-bit physical address. It can be seen that as the number of entries increases, the directory reduction grows beyond what the DGD design achieves in terms of tag reduction alone. In a directory with 16384 entries (which keeps track of 1 MB of private cache blocks), it could reduce the directory by 20 KB. However, it is true that reducing the tag array size is less important to the overall directory size than increasing the associativity.

TABLE 1. PER-ENTRY TAG WIDTH AND PER-BANK TAG FOOTPRINT AS THE PRIVATE CACHE SCALES (128 TILES, 16-WAY DIRECTORY, 64 B BLOCKS, 48-BIT PHYSICAL ADDRESS)

Private cache entries	Tag width (bits)			Per-bank size (KB)		
	Sparse	DGD	CE-DGD	Sparse	DGD	CE-DGD
512	30	35	30	10	10.31	10
1024	29	35	29	19.88	20.62	19.88
2048	28	35	28	39.5	41.25	39.5
4096	27	35	27	78.5	82.5	78.5
16384	25	35	25	310	330	310

6. Adapting to realistic directory mappings

The mapping of cache slices and cache lines in real processors is often more complex than the one proposed here. On Ryzen processors [22], the set and home indices are hashed using a custom XOR function. This hashing does not significantly change the design, as it could be implemented similarly to the proposed one, applying the hash function to the raw index before the access.

On Intel processors [23, 24, 25, 26], however, the mapping is more complex and varies across generations, requiring the entire address (except for the block offset) to be hashed in order to determine its slice and set. This mapping is not publicly documented and differs between processor generations, so it has only been approximated through reverse engineering. In these scenarios, the hash of the address without the block offset is used to determine the home.

Adapting this to a region-based directory is straightforward: the slice is determined by hashing the address without the whole region offset (i.e., using $\log_2(page_size)$ bits for the region offset instead of $\log_2(block_size)$ bits for the block offset). In this way, all blocks within a region are mapped to the same slice and set.

7. Methodology

To evaluate the performance of our proposal, we use a cycle-accurate simulator based on GEMS 2.1 [27], fed with x86 instructions generated by the Sniper simulator [28]. The interconnection network is modeled using Garnet [29]. We simulate a chip multiprocessor with 128 cores, private L1 and L2, and a logically-shared yet physically distributed LLC. Private caches are kept coherent using a MESI cache coherence protocol modeled in the SLICC language. The complete configuration is described in Table 2.

7.1. Simulation Setup

Our evaluation includes benchmarks from the SPLASH-4 (simmedium: `barnes`, `cholesky`, `fft`, `lu_cb`, `lu_ncb`, `water_spatial`; simsmall: `ocean_cp`, `ocean_ncp`, `radiosity`, `radix`, `raytrace`, `volrend`, `water_nsquared`) and PARSEC (simmedium: `blackscholes`, `bodytrack`, `canneal`, `dedup`, `fluidanimate`, `vips`; simsmall: `freqmine`) suites. We account for the variability of parallel applications by repeating each execution 4 times. We also take the parallel region of each application as the Region of Interest. All the directory designs analyzed have a lookup and insertion latency of 5 cycles. Even if an insertion on a ZCache could produce numerous movements that could spend numerous cycles, as described in the original paper [9], the usage of Miss Status Holding Registers (MSHRs) as an insertion buffer to delay the replacement process could remove those extra cycles from the critical path.

TABLE 2. SIMULATED SYSTEM CONFIGURATION

Core parameters	
Type	Out-of-order
Number of cores	128
Issue queue / ROB	205 / 512 entries
Load / store queue	192 / 114 entries
Memory parameters	
Block size	64 B
L1 I/D cache	32 KB, 8-way, 4-cycle
L2 cache	256 KB, 8-way, 11-cycle
LLC	1 MB/tile, 16-way, 18-cycle
Sparse-8×	256 KB/bank, 5-cycle
Directory	512-entry/bank, 5-cycle
Cache organization	L2 inclusive, LLC non-inclusive
Physical address	48 bits
Memory (round trip)	77 cycles
Network parameters	
Topology	2D mesh (16 × 8), HBM-attached
Flit size	16 B
Message size	5 flits (data), 1 flit (control)
Link latency / BW	2 cycles / 1 flit per cycle

7.2. Evaluated Directory Designs

In our evaluation we use the following directory designs:

- **Sparse-8×**: Sparse directory that uses a bit-vector sharer representation with a block-interleaving mapping. We evaluate 16-way directories with 4K entries per bank (8× larger than our standard 512-entry configuration).
- **Sparse**: 16-way sparse directory with a block-interleaving mapping.
- **Samehome$_9$**: 16-way sparse directory with a same-home mapping using a home offset of 9 bits.
- **SameSet$_9$**: 16-way sparse directory with a same-set mapping using a home offset of 9 bits.
- **DGD**: A dual-grain directory derived from the original DGD design, with a 4 KB region size, 8 ways, and a same-home$_{12}$ mapping. As increasing the replacement depth further could affect the critical path of LLC accesses, we use a ZCache replacement depth of two levels. For example, with replacement depth of three levels and 8 ways, the design would require 21 lookups (1 first-level lookup, 4 second-level lookups, 16 third-level lookups); even if pipelined, this could increase directory latency for smaller directory sizes. It implements the ideal distribution described in Section 4.1.

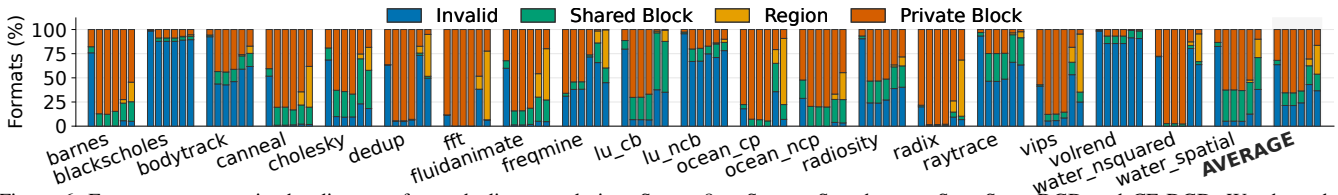


Figure 6. Formats per entry in the directory for each directory design: Sparse-8×, Sparse, Samehome$_9$, SameSet$_9$, DGD and CE-DGD. We show the percentage of each type of entry in the directory averaged from samples taken every 100000 cycles. 'Region' represents a region entry and 'Private Block' represents a block entry that has only one sharer and cannot be merged into a larger region.

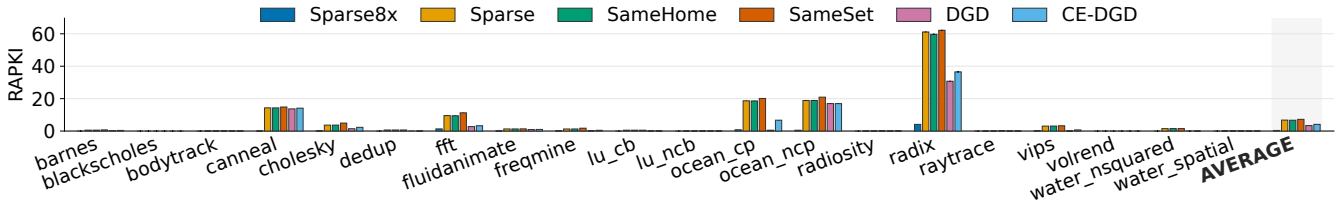


Figure 7. RAPKI in the directory for the different directory designs

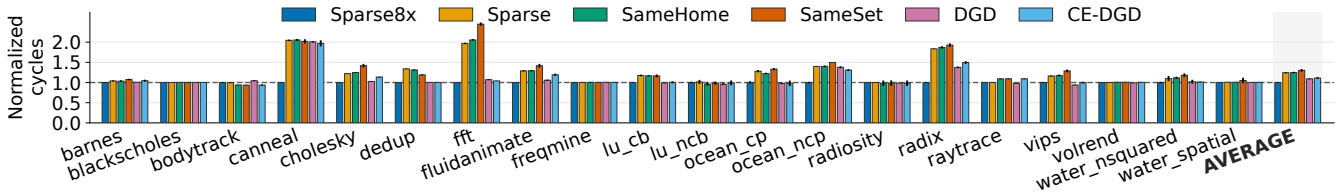


Figure 8. Normalized execution cycles for the different directory designs

- **CE-DGD**: Our cost-effective dual-grain directory with a same-set mapping using a home offset of 9 bits. We evaluate a 16-way directory and a region size of 512 bytes.

For the ablation study we add the following designs:

- **DGD Simple Dist**: Uses the simplified distribution described in Section 4.1 instead of the ideal distribution assumed in DGD.
- **DGD Skewed**: Same as the DGD design but instead of using a ZCache, uses a Skewed cache based on the same lookup as the ZCache.
- **Dual access DGD**: The double-port directory described in Section 4.3 where we search for a block entry in a set and in another set for a region entry.

8. Results

8.1. Analysis of compression factor

We begin by evaluating the extent to which region entries reduce the number of directory entries required to track the blocks held in the private caches. As discussed in the previous sections, grouping multiple blocks under a single region entry allows the directory to track a larger working set with fewer entries. Fig. 6 reports the distribution of entry formats across the SPLASH-4 and PARSEC benchmarks, characterizing both the potential reduction in directory occupancy and, under high directory pressure, the ability of region entries to track multiple blocks within a single entry.

Relative to the Sparse directory, both DGD and CE-DGD require substantially fewer entries to track the same number of blocks on benchmarks that do not saturate the directory, such as `lu_cb`, `bodytrack`, `water_spatial`, and `vips`. This headroom enables a further reduction of the directory size on these workloads without incurring additional latency.

On benchmarks where the directory is fully occupied under DGD or even CE-DGD, namely `fft`, `fluidanimate`, and `ocean_ncp`, a large fraction of entries are represented as region entries, allowing the corresponding blocks to be tracked without triggering directory-induced invalidations. For instance, `fft` exhibits a predominantly per-core private access pattern and therefore benefits the most from region-format compression; on this benchmark, even the Sparse-8× directory reaches roughly 70% occupancy.

Overall, every benchmark employs region entries under the region-based designs. We note, however, that the larger number of region entries observed for CE-DGD does not imply a higher compression ratio than DGD. Because a DGD region entry can represent more blocks than a CE-DGD region entry, DGD can achieve the same coverage with fewer entries. This behavior indicates that increasing the region size could reduce the number of region entries required, as observed in `ocean_cp` and `water_nsquared`, where a single DGD region entry corresponds on average to four CE-DGD region entries. Conversely, coarser regions do not always reduce the entry count: in `freqmine` and `water_spatial`, finer-grained regions yield fewer directory entries, suggesting that 4 KB regions tend to be shared by more than one core, whereas smaller regions remain private.

Nevertheless, the most decisive indicator of whether a region-based directory is beneficial is the degree of reduction in directory-induced invalidations, as can be seen in Fig. 7. Not all benchmarks exhibit a high replacement rate in the baseline configuration; the most representative ones are `canneal`, `cholesky`, `fft`, `ocean_cp`, `ocean_ncp`, and

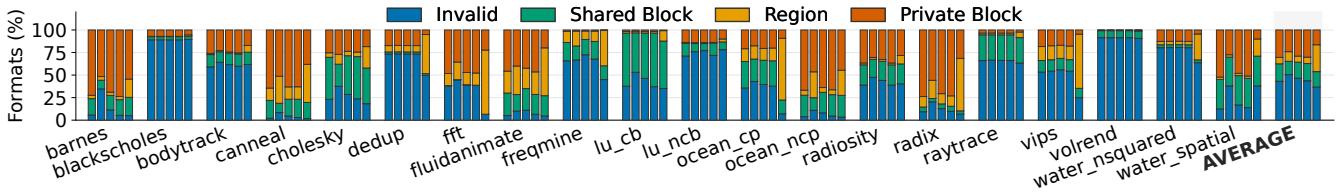


Figure 9. Formats per entry in the directory for each directory design: DGD, DGD Simple Dist, DGD Skewed, Dual access DGD and CE-DGD. We show the percentage of each type of entry in the directory averaged from samples taken every 100000 cycles. 'Region' represents a region entry and 'Private Block' represents a block entry that has only one sharer and cannot be merged into a larger region.

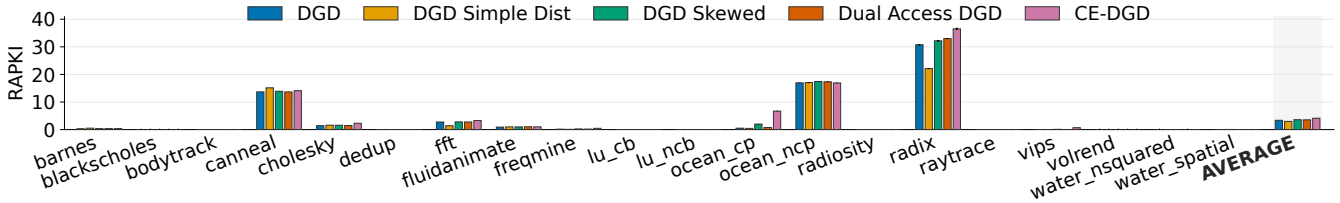


Figure 10. RAPKI in the directory for the ablation study of the DGD design

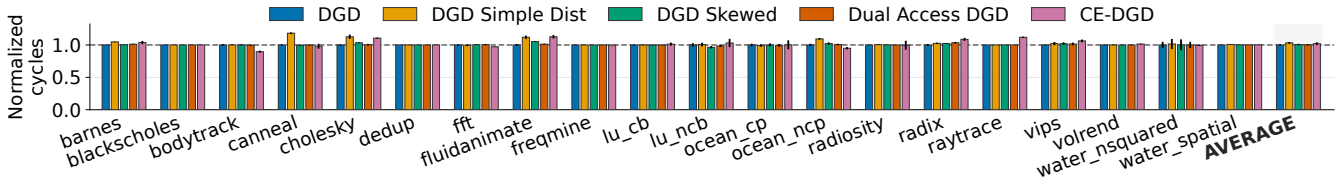


Figure 11. Execution cycles normalized with respect to DGD

`radix`. For `ocean_cp`, `fft`, and `radix`, representing entries as regions reduces directory evictions by approximately one half. In contrast, on `canneal` and `ocean_ncp`, region entries do not reduce the number of replacements.

8.2. Execution time

The central premise of our proposal is that directory-induced invalidations degrade execution time; in the absence of this effect, directory size becomes irrelevant and alternative strategies would be preferable. Fig. 8 confirms a clear correlation between the number of directory replacements and the resulting execution times across the SPLASH-4 and PARSEC benchmarks. Benchmarks whose replacement count does not decrease appreciably, such as `canneal` and `ocean_ncp`, yield only marginal improvements. Conversely, `fft`, `cholesky`, and `ocean_cp` attain execution times comparable to the Sparse-8× directory, demonstrating the effectiveness of region-based directories.

The address mapping likewise influences the performance of each directory design. On average, the mappings described in Section 3 introduce little penalty. Nonetheless, block interleaving outperforms same-set interleaving in some cases, exposing the corresponding degradation of CE-DGD. For example, `raytrace` loses some performance under the same-set mapping relative to block interleaving, although the compression of several block entries into fewer region entries mitigates this loss.

8.3. Ablation study

Examining the entry-format distribution of each region-based design in Fig. 9, we observe that the simplest distribution scheme is among the most adversely affected. In `canneal`, for instance, the abundance of block entries leads to a higher number of evicted shared entries and, consequently, longer execution times. In other benchmarks such as `radix`, a distribution that uses fewer region entries can paradoxically reduce the total number of directory replacements, but, because the remaining replacements affect shared entries, each produces more invalidations and hence is considerably more costly. This evidences that a balanced distribution is needed for performance.

Alternative mapping strategies, such as skewed caches or the Dual-Access DGD directory, provide no average benefit over the baseline DGD design, indicating that DGD is largely insensitive to the choice of mapping. While these variants remain competitive, DGD attains the best overall performance by exploiting the higher effective associativity brought by the ZCache.

CE-DGD preserves a favorable entry distribution despite operating under the least favorable mapping, and it remains competitive across all evaluated designs. On average, it matches the more complex designs, achieving comparable RAPKI (Replaced Addresses per kilo instructions) and execution times as seen in Fig. 10 and Fig. 11.

It is important to remark that in the ablation study, DGD and DGD Simple Dist increase their energy consumption compared with CE-DGD and DGD Skewed on benchmarks like `ocean_ncp` and `canneal` as replacements are common.

9. Related Work

Several proposals in the literature further reduce directory size by targeting different objectives: reducing the number of directory entries, shrinking the size of each entry, or filtering out entries unnecessary for coherence.

To reduce the size of each entry, proposals such as Limited Pointers [2] compress the sharing code by only tracking a subset of the sharers, and broadcast the messages to all the cores when the sharing code overflows. This reduces the entry size optimizing for the common scenario of having a few sharers, but increases the traffic in the interconnection network as well as the latency when the number of sharers is larger. Coarse Vectors [30] track sharers not per core, but per cluster. As the size of clusters increases, the reduction in entry size is greater, but it also increases the traffic in the interconnection network as well as the latency. SPATL [31] identifies common sharing patterns and stores them in a cache. Each block in the directory can reference these sharing patterns, thus reducing the size of the directory, as the number of sharing patterns is much smaller than the number of directory entries, but it relies on the assumption that sharing patterns are stable, which is not always the case and is dependent on the workload and the number of cores involved. Tagless directories [32], on the other hand, use Bloom filters to track which sets of each different core hold each address. It reduces the size of the directory as it does not need to track the tag of each entry and also uses Bloom filters to track the sharers. However, this design does not scale well with many cores and a large number of sets per directory, and uses a special structure, 2D Bloom filters, to group multiple Bloom filters [33] per set into a single one.

Since different workloads exhibit different sharing patterns, modern directory designs focus on supporting multiple entry formats tailored to sharer requirements, enabling more flexible and adaptive coherence structures. Pool directories [34] have different formats of entries, one with limited pointers for addresses with few sharers, and another with segment vectors for addresses with many sharers. Instead of needing a full-bit-vector of sharers, segment vectors address cases when sharers are in clusters. However, this requirement is mostly effective when the operating system (OS) is aware of assigning cores in the same cluster to threads that are good candidates to share data. In other cases, the reduction could be less effective. SCD [35] uses the same concept as the pool directory, but it uses the ZCache to store the different formats of entries. As a ZCache tries to avoid replacements by relocating entries in different ways, it increases the complexity of the directory design. Way combining [36] uses multiple ways for representing sharers of the same address. When enough entries can be used, sharers are represented using limited pointers, but when the number of sharers grows, a more compact coarse vector is used, reducing the precision of the information. This directory design is more scalable, but it uses an imprecise representation of sharers and is dependent on the number of ways of the directory and the number of sharers per address to maintain enough sharer precision.

Filtering coherence reduces directory size by tracking only the entries needed for coherence. Since not all blocks in private caches are shared across cores, there is no need to track the state and sharers of unshared blocks in the directory. By storing only coherence-critical entries, filtering reduces the directory size while maintaining correctness. Some works [37] filter directory entries using TLBs, as blocks accessed only by one core do not need coherence. However, it requires OS recovery methods when an entry is filtered but then needs to keep track of its coherence. Other proposals improve the filtering of the directory by using block-level filtering [38], which allows filtering entries at block level instead of page level but has more overhead.

Multi-grain directories that require changes to the coherence protocol are present in different proposals. RegionScout [39] uses a region-based cache to track unshared regions and uses this information to optimize snoop coherence protocols. However, its importance lies not in reducing directories, but in reducing the number of snoops in a snoop-based coherence protocol. Protozoa [11], on the other hand, uses multi-grain directories but based on smaller block sizes to avoid the problem of false sharing. However, the directory design does not need to be modified, so it does not address the problem described in this article. SS-DGD [10] statically separates the directory into two different caches, one for regions and the other for blocks. However, it relies on an inclusive LLC to reduce the directory size.

Other NUCA mappings like First-Touch or Reactive NUCA, where the home of a cache line is determined by the OS [18, 40], could also be considered. However, we maintain a simpler mapping to simplify our evaluation, as improving the mapping is not within the scope of this work.

10. Conclusion

This work analyzes the trade-off between maximizing performance with a complex directory organization and adopting a simpler, lower-cost implementation. The original DGD design achieves strong performance but relies on complex way-distribution functions and ZCache-based relocation that are difficult to implement in practice.

We propose CE-DGD, a cost-effective alternative that preserves most of DGD's performance while substantially reducing implementation complexity. By mapping region entries to the same set and eliminating ZCache relocation, CE-DGD uses a conventional cache structure with a simpler access path and can reduce tag overhead compared with DGD. Our results show that practical region-based directories can retain most of multi-grain coherence benefits without inheriting the full complexity of DGD.

Acknowledgments

This work has been funded by the Spanish Ministry of Science and Innovation under the 2022 call "Proyectos de Generación de Conocimiento", project DAMAS (PID2022-136315OB-I00, 2023–2027), and by the FSRM/10.13039/100007801 (SCALE project, grant 23039/GERM/25), Spain. Víctor Galindo-Garre is a PhD student funded by Fundación Séneca – Agencia de Ciencia y Tecnología de la Región de Murcia, Spain, under grant 22351/FPI/23.

References

[1] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ser. ISCA '88. Washington, DC, USA: IEEE Computer Society Press, 1988, p. 280–298.

[3] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Comput.*, vol. 27, no. 12, p. 1112–1118, Dec. 1978. [Online]. Available: https://doi.org/10.1109/TC.1978.1675013

[4] Advanced Micro Devices, Inc., "5th Gen AMD EPYC Processor Architecture Overview," AMD, White Paper Doc. ID 70353, Mar. 2025. [Online]. Available: https://docs.amd.com/v/u/en-US/5th-gen-amd-epyc-processor-architecture-white-paper

[5] Intel Corporation, "Intel Xeon 6900/6700/6500-Series Processors with P-Cores: Specification Update," Intel Corporation, Technical Report Doc. ID 835486, Rev. 013US, Feb. 2026. [Online]. Available: https://cdrdv2-public.intel.com/835486/835486_GNR_SU_Rev_013US.pdf

[6] ——, *Leadership Performance with Intel Xeon 6900 P-core Series*, Intel Corporation, Sep. 2024, technical Fact Sheet for Compute-Intensive Workloads. [Online]. Available: https://download.intel.com/newsroom/2024/data-center/Fact-Sheet-Xeon-6-P-Core.pdf

[7] A. Gupta, W.-D. Weber, and T. Mowry, *Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes**. Boston, MA: Springer US, 1992, pp. 167–192. [Online]. Available: https://doi.org/10.1007/978-1-4615-3604-8_9

[8] A. Ros, M. E. Acacio, and J. M. García, "A scalable organization for distributed directories," *J. Syst. Archit.*, vol. 56, no. 2–3, p. 77–87, Feb. 2010. [Online]. Available: https://doi.org/10.1016/j.sysarc.2009.11.006

[9] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 359–370. [Online]. Available: https://doi.org/10.1145/2540708.2540739

[10] Y. Tang, Y. Qiu, Y. Liu, J. Jiao, P. Zhang, and Y. Fan, "SS-DGD: Scalable Short-Entry Dual-Grain Coherence Directoris," in *2022 IEEE 16th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, 2022, pp. 1–3.

[11] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas, "Protozoa: Adaptive granularity cache coherence," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 547–558, 2013.

[12] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 341–350.

[13] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. USA: IEEE Computer Society, 2010, p. 187–198. [Online]. Available: https://doi.org/10.1109/MICRO.2010.20

[14] R. Sampson and T. F. Wenisch, "Zcache skew-ered," *Proc. of WDDD*, 2011.

[15] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 331–342.

[16] ——, "Yet another compressed cache: A low-cost yet effective compressed cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 3, pp. 1–25, 2016.

[17] A. García-Guirado, R. Fernández-Pascual, and J. M. García, "ICCI: In-Cache Coherence Information," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 995–1014, 2015.

[18] A. Ros, M. Cintra, M. E. Acacio, and J. M. García, "Distance-aware round-robin mapping for large NUCA caches," in *2009 International Conference on High Performance Computing (HiPC)*, 2009, pp. 79–88.

[19] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104–117.

[20] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–39, 2017.

[21] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 37–48, 2016.

[22] H. Wang, M. Tang, Q. Wang, K. Xu, and Y. Zhang, "Zenleak: Practical last-level cache side-channel attacks on amd zen processors," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[23] J. McCalpin, "Address hashing in Intel processors," 2018.

[24] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the Intel last-level cache," *Cryptology ePrint Archive*, 2015.

[25] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 48–65.

[26] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world,"

in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[27] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[28] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[29] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 33–42.

[30] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Scalable shared memory multiprocessors*. Springer, 1992, pp. 167–192.

[31] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "SPATL: Honey, I shrunk the coherence directory," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 33–44.

[32] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 423–434.

[33] D. Gupta and S. Batra, "A short survey on bloom filter and its variants," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2017, pp. 1086–1092.

[34] S. Shukla and M. Chaudhuri, "Pool directory: Efficient coherence tracking with dynamic directory allocation in many-core systems," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 2015, pp. 557–564.

[35] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[36] R. Titos-Gil, A. Flores, R. Fernández-Pascual, A. Ros, S. Petit, J. Sahuquillo, and M. E. Acacio, "Way Combination for an Adaptive and Scalable Coherence Directory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2608–2623, 2019.

[37] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 93–104.

[38] B. R. Upadhyay, A. Ros, and N. Murty, "TLB-based block-grain classification of private data," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2020, pp. 122–130.

[39] A. Moshovos, "RegionScout: Exploiting coarse grain sharing in snoop-based coherence," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 234–245.

[40] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 184–195, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1555815.1555779

[41] M. Tomasevic and V. Milutinovic, "A simulation study of snoopy cache coherence protocols," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, vol. i, 1992, pp. 427–436 vol.1.

[42] G. Dewan and P. Biswas, "A snooping cache coherency protocol for hierarchically organized multiprocessors," *Microprocessing and Microprogramming*, vol. 31, no. 1, pp. 105–111, 1991. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0165607408800531

[43] D. R. Carvalho and A. Seznec, "Understanding Cache Compression," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, Jun. 2021. [Online]. Available: https://doi.org/10.1145/3457207

[44] H. T. Olnowich, "Set associative sector cache," Jan. 1985. [Online]. Available: https://patents.google.com/patent/US4493026A/en

[45] S. Demetriades and S. Cho, "Stash directory: A scalable directory for many-core coherence," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 177–188.

[46] M. Chaudhuri, "Zero directory eviction victim: Unbounded coherence directory and core cache isolation," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 277–290.

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.