Way Combination for an Adaptive and Scalable Coherence Directory

Rubén Titos-Gil[1], Antonio Flores[1], Ricardo Fernández-Pascual[1], Alberto Ros[1], Salvador Petit[2], Julio Sahuquillo[2] and Manuel E. Acacio[1]

Abstract—Today, general-purpose commercial multicores approaching one hundred cores are already a reality and even thousand core chips are being prototyped. Maintaining coherence across such a high number of cores in these manycore architectures requires careful design of the coherence directory used to keep track of current locations of the memory blocks at the private cache level. In this work we propose a novel organization for the coherence directory that builds on the brand-new concept of *way combining*. Particularly, our proposal employs just one pointer per entry, which is optimal for the common case of having just one sharer. For those addresses that require more than one pointer, we have observed that in the majority of cases *extra* pointers could be taken from other empty ways in the same set. Thus, our proposal minimizes the storage overheads without losing the flexibility to adapt to several sharing degrees and without the complexities of other previously proposed techniques. Through detailed simulations of a 128-core architecture, we show that the way-combining directory closely approaches the performance of a non-scalable bit-vector sparse directory, and beats other scalable state-of-the-art proposals.

Index Terms—Cache coherence, sparse directory, way combining, scalability, coverage, bit vector, limited pointers, execution time, network traffic.

1 Introduction and Motivation

Current mainstream multicore architectures implement the shared-memory abstraction as the low-level programming paradigm, and this trend is not likely to change in the foreseeable future [1]. Communication between cores in these devices occurs by writing to and reading from shared memory, while one or more levels of private caches in each core ensure low-latency memory accesses and reduced pressure on shared resources (interconnection network and shared cache levels). A cache coherence protocol implemented in hardware is responsible for preventing cores from observing multiple versions of the same data, thus making private caches functionally invisible to software [2].

Today, general-purpose multicores with close to one hundred cores are becoming commercially available, such as Intel's 72-core x86 Knights Landing MIC [3]. Meanwhile, researchers are already prototyping thousand core chips, like the KiloCore chip developed at UC Davis [4]. Maintaining coherence across hundreds of cores in these manycore architectures requires careful design of the coherence directory used to keep track of current locations of the memory blocks at the private cache level. Duplicate tag directories employed in some first-generation multicores [5] are plainly and simply unfeasible for manycores, since their associativity grows with the number of cores. Contrarily, sparse directories [6] maintain an explicit sharer list per entry and can be organized as typical associative caches, allowing for more scalable implementations. Thus, recent proposals have built on sparse directories [7], [8], [9], [10], [11], [12].

Two aspects determine the area requirements of a sparse directory [13]: The *total number of entries* and the *number of bits of each entry*. The former determines the maximum number of addresses that the directory can contain in a given moment, and therefore has a direct effect on the amount of different memory blocks that can be stored at the private cache level. The term *coverage* is typically used to indicate the number of directory entries with respect to the total number of entries in the last level of private cache. Coverage shortage leads to increased miss rates in private caches due to directory invalidations, hence affecting performance. Multiprogrammed workloads consisting of sequential programs place the most stringent demands on the coverage of a sparse directory, requiring at least as many entries as the sum of all entries in the last level of private caches, to allow all such cache entries to be used at the same time. Previous works (such as [9]) have shown also that in general 100%-coverage is enough in most cases to eliminate nearly all invalidations due to directory evictions if enough associativity is provided.

Whereas coverage does not depend on the number of cores and therefore it is not a scalability hurdle, the amount of bits of each directory entry poses severe limits to system scaling. The size of each directory entry depends fundamentally on how it stores the sharers list for the associated address. To be scalable, directory implementations need to ensure that the number of bits per tracked sharer scales gracefully (i.e. remaining constant or increasing very slowly) [9]. Bit vectors are known to be non-scalable, since their size increases linearly with the number of cores, thus making them unfeasible for large core counts. Alternative representations such as limited pointers [14], [15] or compressed sharing codes [6], [16] curb directory memory overhead. Unfortunately, the improved scalability comes at the cost of increasing either the number of messages per coherence event or the miss rates at the private cache levels. For instance, the loss of precision introduced by coarse bit-vectors [6] leads to more invalidation messages per write, while pointer recycling policies [14] must invalidate privately cached

• [1]*Dept. Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30100 Murcia (SPAIN)*
E-mail: {rtitos, aflores, rfernandez, aros, meacacio}@ditec.um.es
[2]*Dept. Informática de Sistemas y Computadores, Universitat Politècnica de València, 46022 Valencia (SPAIN)*
E-mail: {spetit, jsahuqui}@disca.upv.es

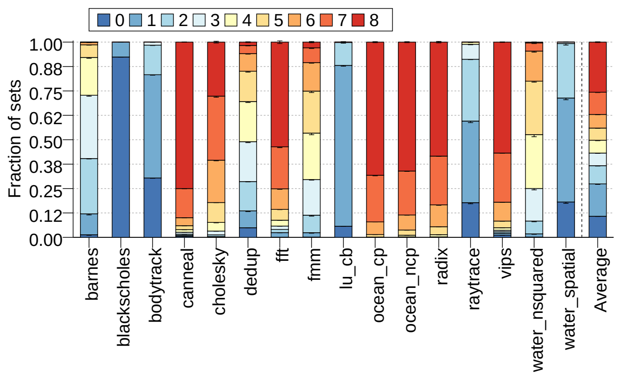


Fig. 1: Directory occupancy per set: average fraction of sets with a given number of occupied entries (ways) in a 100% coverage 8-way sparse directory with bit-vector sharing code for 256 cores (1 sample every 100000 cycles).

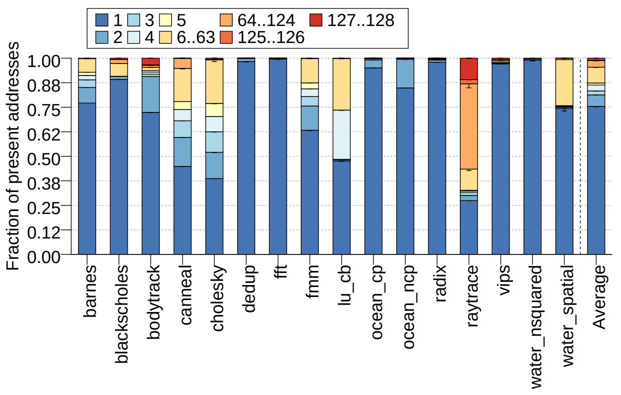


Fig. 2: Sharers per directory entry: average fraction of present addresses with a given number of sharers in a 100% coverage 8-way sparse directory with bit-vector sharing code for 256 cores (1 sample every 100000 cycles).

blocks every time a pointer is reused for a new sharer. At the end, both *extra* coherence messages and increased miss rates result into performance degradation.

It is also well-known that the degree of sharing varies across memory blocks and over time within applications, so that there is no optimal sharers list organization for all cases. Ideally, each directory entry should have enough flexibility to adapt to different situations. Several previous works show that a significant fraction of the directory entries (approaching 90% in some cases) track private blocks, for which a single pointer would suffice. Furthermore, amongst entries tracking shared blocks, most of them have a very small number of sharers (two or three). The remaining very few entries have many sharers, yet its number does not grow with system size [12]. Moreover, virtually all directory entries would track private blocks when sequential workloads are executed in multiprogramming.

This way, a sparse directory designed for the common case should have as many entries as the last level of private caches (with the same or higher associativity), with each entry consisting of a single pointer. Though this design would fit perfectly well to the requirements of sequential workloads in a multiprogrammed environment, when multithreaded applications come into play, the shortage of bits in each directory entry could have catastrophic effects on performance. However, when multithreaded applications are executed, a significant number of directory sets are not fully occupied (i.e. there are free ways in the set) as a consequence of shared blocks appearing in the L2 caches. For the benchmarks considered in this work, Figure 1 shows that sets are on average at half their maximum occupation, and Figure 2 depicts the number of sharers tracked by each entry (refer to Section 3 for details). Interestingly, most of those applications that exhibit high occupancy in Figure 1 (such as Fft, Radix or Ocean_cp) have just one sharer per entry in almost all entries. This observation is not new as it is what motivates previous approaches that use multiple entry formats to store sharing information [9] [17]. We however exploit it differently than previously done. Particularly, we propose that overflowed directory entries in a particular set can expand to the free ways in that set.

Taking into account these observations, in this work we propose a novel sparse directory architecture that builds on the following design principles:

- It should be *designed for the common case*. Considering that the degree of sharing for most addresses is low (one or two), our proposal employs just one pointer per entry.
- It should *adapt to changing sharing degrees*. Though a single pointer suffices for most addresses, there are others which require additional storage to track their sharers list. To handle those with the minimum loss of precision, we leverage the available ways that often exist in the same cache set to allocate additional sharing code storage, giving birth to the concept of *way combining*. This enables flexible resource assignment within a set, making each set of the sparse directory appear as a pool of entries which are dynamically allocated on demand among the addresses mapped to that set.
- It should *entail as lower complexity as possible*. Way combining comes with minimal cost as it avoids the complexity introduced by other proposals [7] [9] [17]. Our proposal builds atop traditional sparse directories, relies on existing replacement algorithms, and does not increase the complexity of directory operations. Of course, it is not as flexible as SCD [9], but we show that extra flexibility enabled by SCD barely has any positive influence on final performance.
- It should keep *directory memory overhead as low as possible*. Our proposal has a lower memory overhead than SCD, which we consider the most scalable directory proposal to date, and this overhead grows more slowly with the number of cores.
- It should approach as much as possible the performance of the non-scalable bit-vector sparse directory. Our proposal reaches this objective (just 2% overhead on average is observed) at the same time that improves over the previously proposed SCD directory.

The rest of the manuscript is organized as follows. We present our proposed directory architecture in Section 2. Section 3 describes our simulation environment and detailed results are shown and analyzed in Section 4. Some important related works are then discussed in Section 5, and finally, Section 6 contains the main conclusions of this work.

2 The Way-Combining Directory

2.1 General Overview

The *way-combining* sparse directory (henceforth, *WC-dir*) stores sharing information about block addresses that are kept at the private levels of the on-chip cache hierarchy typically found in a

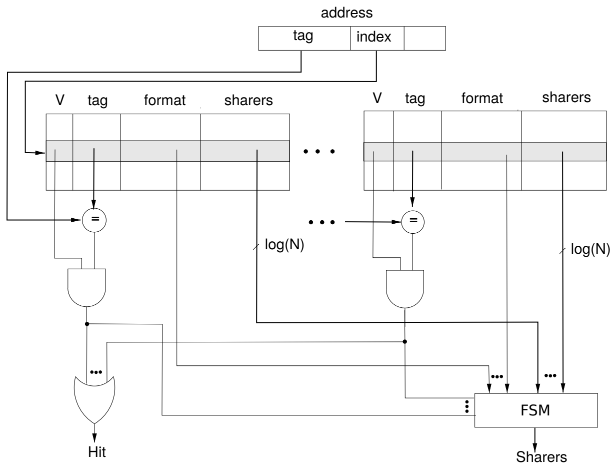


Fig. 3: Implementation of the Way-Combining Directory.

manycore chip multiprocessor. The structure of WC-dir is nearly identical to that of a traditional set-associative directory cache. Figure 3 gives a simple overview of a circuit for obtaining the list of sharers (see Section 2.3 for further details).

Each address is unequivocally mapped to a set in the cache, and the sharing information, if present, may be stored in any set entry. However, unlike a conventional directory cache, WC-dir allows *multiple* entries of the set to be allocated to the same address, so that an access to WC-dir can result in zero, one or more *tag hits*. In the latter case, the sharing information stored in the matching entries is *combined* to produce the list of sharers for the requested address. As depicted in Figure 3, WC-dir replaces the N-to-1 multiplexer typically found in an N-way set-associative cache (which selects the data from the matching entry) with a combinational unit named *FSM* (from Finite State Machine) whose purpose is to merge the sharing information from all the matching entries.

Our design is based on the observation that most memory blocks have only a handful of sharers, most often just one. The dominance of entries with a single sharer (i.e., tracking private data) comes at no surprise in single-threaded multiprogram workloads Nevertheless, in multi-threaded or parallel applications the majority of the directory entries also track private blocks. Furthermore, the common case for shared blocks is that a large fraction of them are only held by two or three sharers. This means that traditional sparse directories that use full bit-vectors to encode sharers clearly make a poor utilization of the area dedicated to storing sharing information.

Another important fact to understand our design is that when two or more private caches hold copies of a block, only one entry needs to be allocated in the directory. That means that in a 100% coverage directory there has to be a free directory entry for every sharer but the first one of every address present at the private cache level. WC-dir can take advantage of those empty entries when they happen to be in the same cache set as addresses whose sharing information does not fit in a single entry.

To take advantage of these observations in a simple design, WC-dir allows entries of the same cache set with the same tag (i.e., referring to the same cache block) to be combined. The sharing information of each block can be encoded in one or more entries of the same set by using either pointers or coarse bit-vectors. For this purpose, two formats, namely pointer and coarse vector, are employed to track the set of sharers of a given block. The format of each entry is encoded with an additional *format field*. Entries in pointer format (assumed to be set to '1' in the example) contain a pointer to a sharer, while entries in coarse bit-vector format [6] (assumed to be set to '0'), contain a portion of the coarse vector of sharers. More precisely, in the coarse bit-vector format, each bit of an entry represents a set of nodes (thus, this representation results in loss of precision). If a bit is set to 1, it means that a copy of the block is maintained in the private caches of one or more of the represented nodes, while if a bit is reset, none of the them holds a copy.

The list of sharers is jointly stored by all combined entries and can be decoded using the referred FSM logic. The ability of WC-dir to combine entries in the same set is independent of the format employed to track the sharers. In fact, the format in which the sharing code is stored for a given address may change over time, depending on the number of entries that can be allocated to the address.

Every time a new block address is inserted into the directory, the pointer format is used by default for the new allocated entry. Subsequent sharers of the same block are also added in pointer format, provided that there are free entries in the directory. However, when directory resources become insufficient to maintain exact sharing information, the amount of directory storage dedicated to specific addresses is dynamically reduced at runtime. This is done as an attempt to maximize directory utilization and precision while keeping low area overhead and operation complexity.

The addresses whose directory storage is reduced are selected as follows. If an address in coarse format exists in a full set (i.e. a set where all the entries are valid), WC-dir makes room by decreasing the number of occupied entries (hence reducing precision). Otherwise, WC-dir entries allocated to an address in pointer format are switched to coarse format in order to make room.

Since evicting an address from the directory results in invalidations in private caches, that may later harm performance by causing additional misses, WC-dir always tries to minimize evictions at the cost of reducing the precision of the sharing code. Thus, evictions only occur when a new address is inserted into a full set where each entry is allocated to a different address, following a typical LRU replacement algorithm to select the victim.

Finally, note that the implementation complexity of WC-dir would be lower than in other proposals such as SCD, since in WC-dir all operations involve a single set.

2.2 Working Example

To illustrate the behavioral aspects of WC-dir, Figure 4 shows the evolution of a 4-way set associative WC-dir for 256 nodes. Each sharer field consists of 8 bits that, in pointer format, can be combined to point up to four sharers (one per cache way) or, in coarse format, to compose a 32-bit (4×8) sharer vector where each bit represents 8 (256/32) nodes.

Figure 4 (a) shows the set containing two addresses *addrA* and *addrB*, both in pointer format and each with a single sharer. New sharers can be added to an existing address by allocating available entries in the set, as depicted in Figure 4 (b). When all entries in a set are allocated (either to the same or different addresses) using the pointer format, no sharer can be inserted into the directory without first taking action to make room in the set.

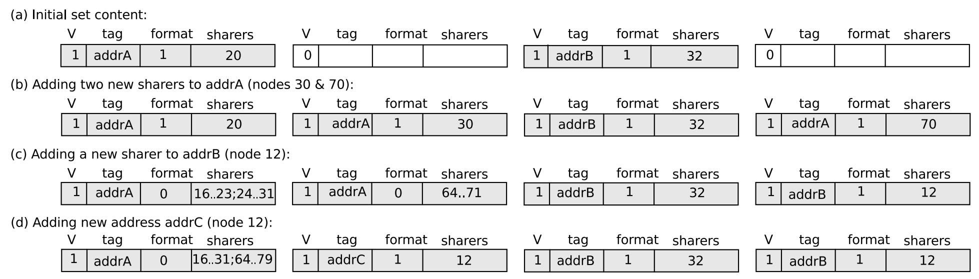


Fig. 4: WC-dir: Example of operation.

Figure 4 (c) shows how before inserting a new sharer for *addrB*, *addrA* must switch from pointer format over three entries to coarse vector format over two entries, thus releasing one of its entries (note that no address is evicted). Though in this example there is only one candidate, in practice there are several heuristics that could be employed to select the victim amongst the candidate addresses. In this work, WC-dir opts for a simple LRU algorithm, although other approaches could be used. Also, those candidates whose sharing code is already stored in coarse format are always chosen over those in pointer format, in order to keep precise sharing codes for as many addresses as possible (as in Figure 4 (d)).

2.3 Low-level implementation details

We assume that any number of entries can be combined if they are in pointer format, while this number must be a power of two to be combined when they are in coarse format. That is, one, two, four or eight entries can be combined in coarse format in an 8-way set-associative cache. If 8 entries are being combined (i.e. 64 bits in total), then the *granularity* of the coarse bit-vector representation is equal to 4, since each bit represents four nodes ($NPROCS/64$). Notice that the coarsest granularity is 32 nodes represented by a single bit, when a single entry is used to represent all the nodes.

The different coarse granularities are checked by the auxiliary hardware logic at runtime. This is done when the tag of the block is being looked up. At this time, the number of entries matching the same tag (i.e. referring to the same block address) is counted. To conserve energy, we assume that the data array is accessed after tags, as typically done in second and low level caches.

The illustrate the simple hardware used by our approach, next, we depict some circuit examples involved in the three major actions carried out by WC-dir: i) adding a new sharer to an entry, ii) making room for a new sharer in a full set, and iii) reading the list of sharers.

Adding a new sharer to an entry. To add a sharer a circuit similar to the depicted in Figure 5 can be applied. For entries in coarse format, the three bits from the sharer pointer ($PTR_{X..X-2}$) that are used to index the bit representing the node in the coarse bit-vector are selected depending on the granularity, which can be obtained from the number of combined ways (*n-way*). The resulting index is decoded and *ORed* with the coarse bit-vector to obtain the updated vector with the new sharer. On the other hand, in case the target address is being codified in pointer format, then PTR is just written to the entry. Note that in this case a free entry is required. Otherwise, more room must be made for the new sharer.

Making room for a new sharer. When additional space is required to store a new sharer, the design can choose between either increasing the granularity of a sharing code in coarse format, or moving from pointer to coarse format. Since the approach pursues to achieve the highest precision, the former case is always done incrementally as additional room is needed. That is, the number of ways devoted to a sharing code in coarse format is reduced to the immediately lower power of two (e.g. from 4 to 2 entries, or from 8 to 4 entries). Figure 6a presents an example of a circuit performing such an increase of granularity. In the figure, a portion of 16 bits (2 entries) from a coarse bit-vector are ORed to obtain 8 bits representing the same nodes but with lower precision. Note that since the amount of storage is halved, one entry (the left one) is released (the valid bit is set to '0').

Regarding to moving from pointer to coarse format, Figure 6b shows a possible implementation. In this example, two pointers are combined in the same entry in coarse format. Similarly to Figure 5, the bits of each pointer that are used to index the target coarse bit-vector depend of the number of ways allocated to it.

Reading the list of sharers. To obtain the list of sharers, those ways of the set whose tag match the target address must be read and fed to the FSM, which produces a set of pointers to the nodes involved in the coherence action. The way to obtain the set of pointers depends on the format. Figure 7 depicts two examples. In Figure 7a, the FSM is fed with the 8 bits of one entry, where each bit represents 32 nodes (i.e. the coarsest granularity), while in Figure 7b a sharing code with two pointers is read.

3 Evaluation Methodology

We evaluate the performance of different cache coherence directories using the GEMS 2.1 simulator [18]. GEMS is fed with information gathered by a PIN tool [19], which offers detailed

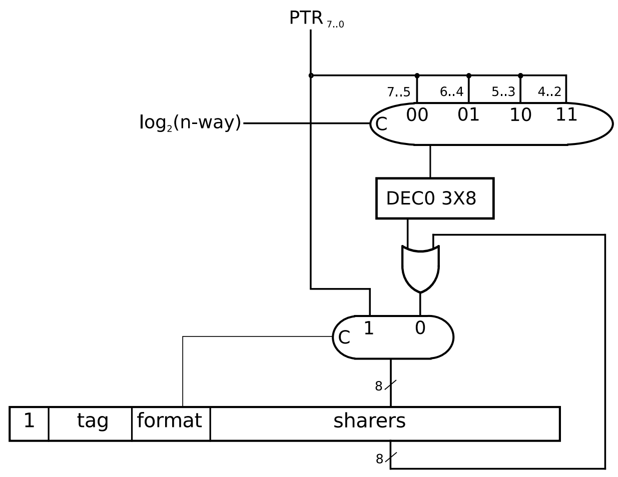
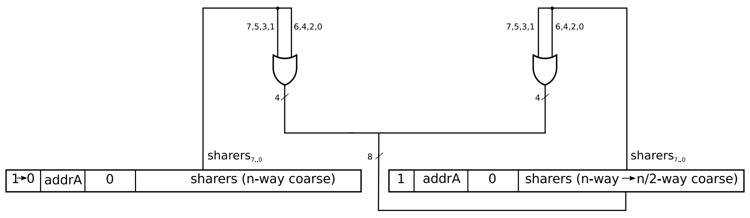
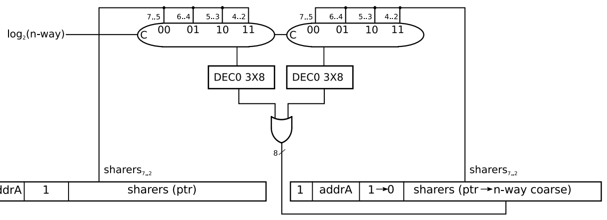


Fig. 5: Circuit for adding a sharer.

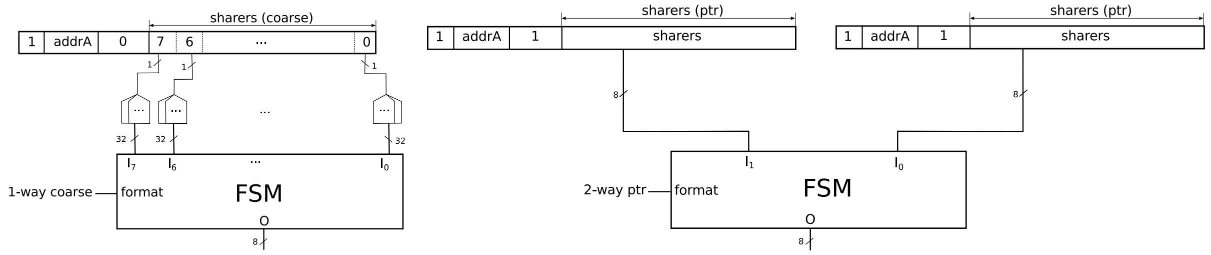

7,5,3,1
6,4,2,0
4
sharers7..0
8
1→0 addrA 0 sharers (n-way coarse)
1 addrA 0 sharers (n-way→n/2-way coarse)

(a) n-way to n/2-way coarse


log2(n-way)
C 00 01 10 11
DEC0 3X8
8
sharers7..2
1→0 addrA 1 sharers (ptr)
1 addrA 1→0 sharers (ptr→n-way coarse)

(b) 2 pointers to n-way coarse

Fig. 6: Examples of changing the format.



(a) 1-way coarse format

(b) 2-way pointer format

Fig. 7: Examples of reading the list of sharers encoded in different formats.

TABLE 1: System parameters.

Memory parameters	
Block size	64 bytes
L1 cache (data & instr.)	32 KiB, 4 ways
L1 access latency	1 cycle
L2 cache (data & instr.)	128 KiB, 8 ways
L2 access latency	10 cycles
L3 cache (shared)	1024 KiB/tile, 32 ways
L3 access latency	20 cycles
Cache organization	L2 inclusive, L3 non-inclusive
Directory size (SCD75)	1536 entries, 3 ways (75% coverage)
Directory size (SCD)	2048 entries, 4 ways (100% coverage)
Directory size (rest)	2048 entries, 8 ways (100% coverage)
Directory latency	5 cycles
Physical address size	48 bits
Memory access time	200 cycles
Network parameters	
Topology and Routing	2-D mesh (8×8), X-Y
Flit size	16 bytes
Message size	5 flits (data), 1 flit (control)
Link time	2 cycles
Bandwidth	1 flit per cycle

TABLE 2: Benchmarks.

SPLASH-3	
Barnes	16K particles, timestep = 0.25, tolerance = 1.0
Cholesky	13992×13992, NZ=316740
Fft	2^{20} total complex data points
Fmm	16K particles, timestep = 5
Lu_cb	512×512 matrix, block = 16
Ocean_cp	514×514 grid, distance = 20000, timestep = 28800
Ocean_ncp	514×514 grid, distance = 20000, timestep = 28800
Radix	4M keys, radix = 4K
Raytrace	Balls4, antialiasing with 2 subpixels
Water_nsqared	8^3 molecules, timestep = 3
Water_spatial	15^3 molecules, timestep = 3
PARSEC 3.0	
Blackscholes	4096 options
Bodytrack	4 cameras, 1 frame, 1000 particles, 5 annealing layers
Canneal	5000 swaps per temperature step, 2000° start temperature, 200000 netlist elements
Dedup	31 MB
Vips	2336×2336 pixels

information about the instructions executed, memory references, and syncronization primitives as is the standard methodology for large-scale system simulations [20]. We model the interconnection network with Garnet [21]. The simulated architecture corresponds to a single chip multiprocessor (*tiled*-CMP) with 128 cores (one per tile). All evaluated configurations implement local caches with MESI states. The most relevant simulation parameters are shown in Table 1.

We evaluate five configurations for the coherence directory that we name BV, LP1, SCD, SCD75 and WC1. BV employs a sparse directory using non-scalable bit-vectors in each directory entry as the sharing code. LP1 is an implementation of Dir_iCV [6] which uses a limited pointer scheme in which the sharing information is stored as a single pointer in the case of private blocks or as a coarse bit-vector when several sharers are found. SCD is an implementation of the SCD architecture [9] using a 4-way z-cache that explores three levels when finding a replacement candidate (which means that it is roughly equivalent to a 52-way associative cache). SCD75 is a different configuration of SCD with only 75% coverage whose area requirements are closer to those of

LP1 and WC1, since it uses a 3-way z-cache that explores four levels (roughly equivalent to a 45-way cache). Finally, WC1 is an implementation of WC-dir that uses 1-pointer entries. BV and LP1 use silent replacements of shared blocks (no notification is sent to the directory in case of eviction of a clean shared block) and WC1, SCD and SCD75 use noisy replacements (a notification is always sent to the directory upon eviction). We have evaluated both options for each configuration and selected the best shared block replacement technique in terms of execution time for each case.

Our simulations consider representative applications from PARSEC 3.0 [22] and SPLASH-3 [23] (see Table 2). We have included as many benchmarks as we have been able. We have excluded only those benchmarks that we could not scale up to 128 cores (i.e. execution time with 128 threads is smaller than with 64 threads) and Freqmine, which uses OpenMP and cannot be ported to our simulations infrastructure. Input set sizes have been fixed considering resulting simulation times. The resulting set of benchmarks contains applications exhibiting varying behaviors and sharing patterns, with an average L2 miss rate of 64% All the results correspond to the parallel part of the applications and we have accounted for the variability of parallel applications.

4 Evaluation

Table 3 shows the amount of memory required to implement each of the directory structures considered in this work. The data for LP1 has been omitted because it is identical to that of WC1. In addition to the sizes for 128-core systems, which are considered in the rest of this section, the data of smaller and bigger systems are also shown to illustrate the scalability of the different proposals. For each tile, the BV directory requires more than 39 KiB to support a 128 KiB last private cache, while WC1 and LP1 require only 9.3 KiB, thanks to the much smaller sharing code. SCD with the same coverage as the rest requires significantly more area than LP1 and WC1 both because the sharing code needs more bits and because the tags required by the z-cache are larger. Even reducing the coverage of SCD to 75%, it still requires more memory than LP1 and WC1 for 128 or more nodes. Moreover, if we look at how the size (per tile) of each directory scales with the number of nodes, we can see that only LP1 and WC1 keep their overhead constant. This happens because the tag size is reduced at the same rate as the sharing code size increases (i.e., logarithmically). The size of the sharing code of BV grows much faster, to the point that the directory would need more area than the tracked caches for a system with 512 nodes or more, making it non-scalable. SCD scales much better than BV but worse than LP1 and WC1. This is because its sharing code size grows faster than WC1 and LP1's one (as the square root of the number of nodes) and its tag size remains constant.

The larger memory requirements imply more area, and thus, higher static energy consumption for the directory. Hence, for core counts larger than 64, WC1 (and LP1) is the scheme that would consume less static energy, being the reduction with respect to the other approaches more notable as the core count increases.

Each directory design makes use of its allocated resources in a different way to store the sharing information of the addresses present in the private caches. This will determine how easy it is to access and update that information and how precise it is. In some cases, a directory design will reduce the precision of the stored information (always by storing a superset of the actual sharer set) at the cost of more invalidation traffic. Figure 8 shows the average precision per address stored in the directory during the whole execution of the applications. Both BV and SCD achieve perfect precision, although SCD does that with much fewer resources. LP1 and WC1 have lower precision, but we can see that way combining allows WC1 to improve the precision of the information stored in the directory with respect to LP1, which needs the same amount of resources. As expected, the improvement is more marked in those benchmarks that have fewer occupied entries per set (see Figure 1). Note, however, that not all tracked blocks will be necessarily written (read-only blocks), and some blocks will be updated more frequently than others. Thus, approaching perfect precision is generally important but in some cases it could come without any benefits.

Figure 9 plots the number of directory replacements per instruction. As already explained in Section 2, WC1 is designed so that it can hold exactly the same number of addresses as BV and LP1. Obviously, WC1 stores these addresses with increased precision over LP1. To ensure this, WC1 only combines entries when empty ways are found in a particular set. This way, WC1 never allocates new entries to an address at the expense of expelling another address in the same set. In that case, the first address is transitioned into the coarse vector representation. We can see that WC1 has fewer directory replacements than BV and almost as many as SCD. This is because, as explained in Section 3, both WC1 and SCD are using noisy replacements of shared blocks while BV is using silent replacements, and noisy replacements enable the deallocation of entries for addresses evicted by all sharers, reducing the directory occupancy. Regarding SCD, we can see that reducing the size of the z-cache to 75% (SCD75) increases dramatically the number of directory replacements. This is because L2 caches are usually almost full and a directory with 75% coverage, even when SCD provides increased flexibility in allocating directory entries, is unable to keep all the addresses which could be stored at the L2 caches (i.e., L2 cache resources are wasted). Interestingly, we can also notice that in some cases (i.e., Canneal, Ocean_cp, Ocean_nc and Vips), SCD with 100% coverage results into increased directory replacements with respect to WC1. This is because SCD uses one extra entry to store indexing information for blocks with several sharers, thus reducing the total effective capacity of its cache. Figure 10 shows the number of L2 cache replacements per instruction, where we can see that SCD75 reduces the number of L2 replacements with respect to the rest because its reduced coverage often forces the invalidation of many lines before the sets get full, wasting space in the caches.

Figure 11 shows the average L2 miss latency split in five components: the time that the miss spends in L2 before being issued (*At_L2*), the time that the request takes to arrive to L3 (*To_L3*), the time that it spends waiting before being attended (*At_L3*), the time spent accessing memory (*Main_memory*) and the time until the data and all acknowledgments arrive to the requestor (To_L2). We observe that LP1 and WC1 increase the *To_L1* time for a few benchmarks (i.e., Barnes, Canneal, Cholesky, Fmm, Ocean_cp, Ocean_ncp, Water_nsqared and Water_spatial). This is because these configurations generally send more invalidations on write misses due to the lack of precision of their sharing information, as can be seen in Figure 12. But the increase incurred by WC1 is much smaller than that of LP1 on most benchmarks, becoming practically none in some of them (e.g., Barnes, Fmm and Ocean_cp).

TABLE 3: Directory size and overhead for different configurations (LP1 sizes are identical to WC1).

Nodes	64				128				256				512				1024			
Directory	BV	SCD	SCD75	WC1	BV	SCD	SCD75	WC1	BV	SCD	SCD75	WC1	BV	SCD	SCD75	WC1	BV	SCD	SCD75	WC1
Tag (bits)	28	36	36	28	27	35	35	27	26	34	34	26	25	33	33	25	24	32	32	24
Sharing Code (bits)	64	11	11	7	128	16	16	8	256	20	20	9	512	28	28	10	1024	37	37	11
Size / Tile (KiB)	23.5	12.3	9.2	9.3	39.3	13.3	9.9	9.3	71.0	14.0	10.5	9.3	134.8	15.8	11.8	9.3	262.5	17.8	13.3	9.3
% over L2	17.2	8.9	6.7	6.8	28.6	9.7	7.3	6.8	51.8	10.2	7.7	6.8	98.4	11.5	8.6	6.8	191.6	13.0	9.7	6.8

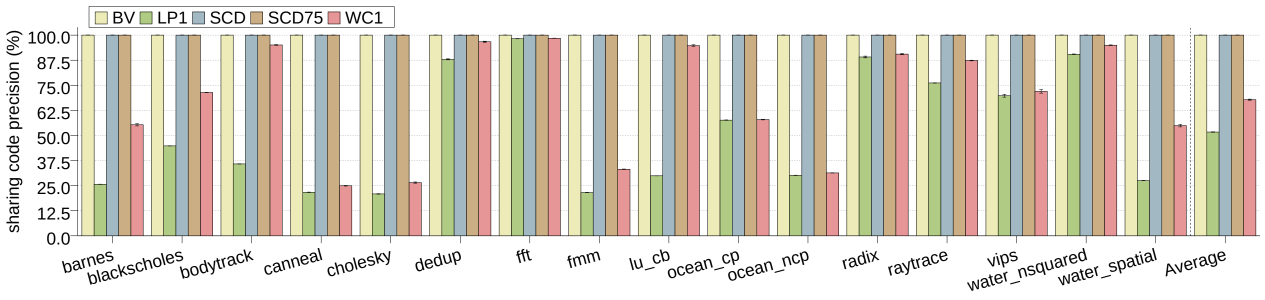


Fig. 8: Precision per address measured as the average for each address of the ratios between the actual number of sharers and the number of sharers encoded in the directory. The directory is sampled every 100000 cycles.

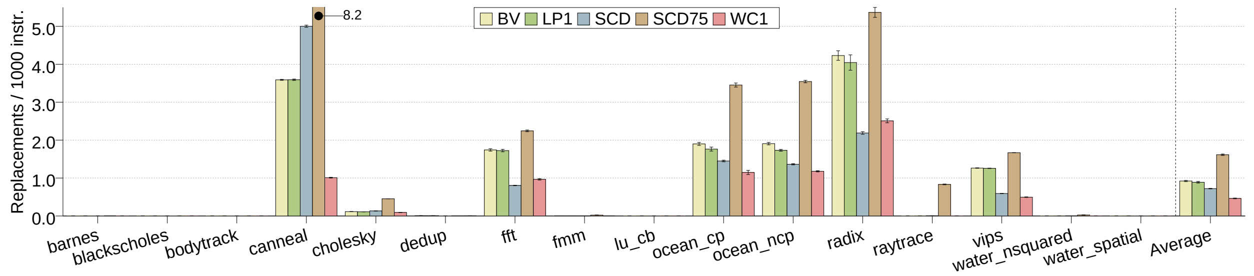


Fig. 9: Directory replacements per instruction.

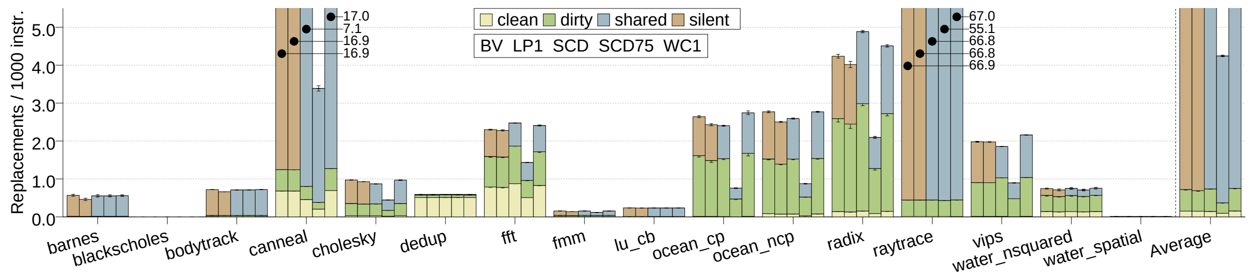


Fig. 10: L2 cache replacements per instruction.

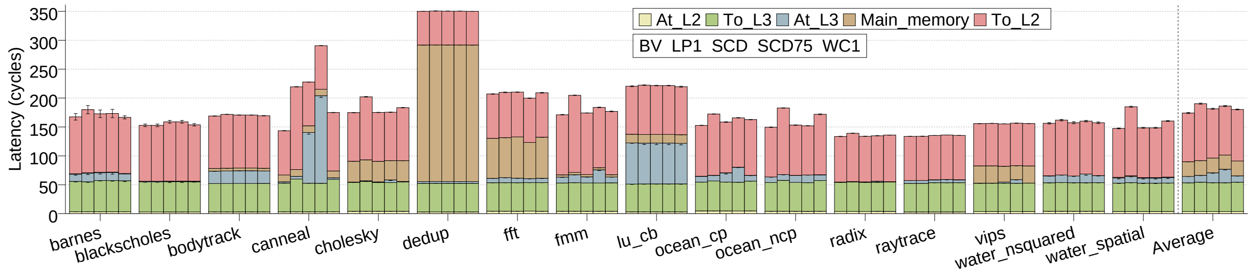


Fig. 11: L2 miss latency.

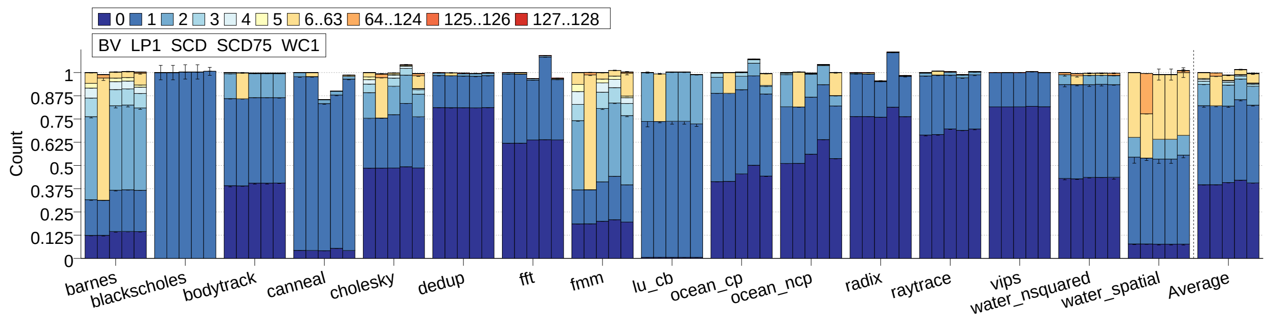


Fig. 12: Frequency of each number of sharers invalidated per L2 write miss.

Also, we can see in Figure 11 that the *At_L3* time of SCD and SCD75 increases for some benchmarks (i.e., Canneal, Ocean_cp and Vips). We have found that this is caused by extra directory replacements due to having to use more than one entry per address (as already commented on) and because replacements in SCD are more expensive than in any of the other configurations. Particularly, on a replacement in SCD, more accesses to the z-cache are necessary to move entries to make room, keeping the directory busy for more time. WC1, on the other hand, prefers to dynamically reduce the precision of the sharer set of some addresses rather that evict them. The results show that the extra traffic and latency due to the extra invalidations is not so bad as the extra latency in SCD due to the directory replacements.

The most direct effect of the lack of precision of the directory information is that unnecessary invalidation messages are sent upon write misses, as shown in Figure 12, and upon directory replacements. These extra messages can have in some cases significant effect in the total network traffic, as shown in Figure 13. Here again we see that the increased precision afforded by way combining allows WC1 to have much lower traffic than LP1, although it is still higher than BV's and SCD's. For most benchmarks, the increase in traffic does not have an important effect on miss latency, as already seen in Figure 11, and hence will not affect the execution time in a significant extent. Interestingly, though SCD reaches perfect precision, the difference in average traffic regarding WC1 is just 10%, even though SCD has significantly larger area requirements. In this figure we show, in addition to the global average, the average of a selection of those benchmarks that have more L2 replacements (Canneal, Fft, Ocean_cp, Ocean_ncp, Radix, Raytrace and Vips). We can see that the traffic increase of WC1 for these benchmarks is slightly higher, but still lower than LP1.

Dynamic energy consumption is fundamentally affected by the differences in network traffic. First, the dynamic energy consumption of the interconnection network is proportional to its traffic load and has been reported to constitute a significant fraction of the total energy budget [24]. Second, unnecessary invalidation messages increase the number of snoops in the private caches. These snoops, however, are much less frequent than the accesses from the local processor, and therefore, the difference on dynamic energy consumption is minimal.

Figure 14 shows the relative increase in normalized execution time for each directory structure. First, it proves that reducing the coverage of SCD to 75%, to make its memory requirements similar to LP1's and WC1's has a very negative effect in many benchmarks (e.g., Canneal or Ocean_cp), such that on average SCD75 performs worse than LP1. SCD with full coverage achieves an execution time that is less than 5% slower on average than BV, and it even outperforms it in some cases (e.g., Fft and Radix). The latter is due to the increased effective associativity provided by the z-cache used in SCD, that eliminates some conflict misses appearing in BV. Finally, WC1 average overhead with respect to BV is just 2%, thus being the configuration that closest approaches the performance of the non-scalable BV. If we look only at those benchmarks with many L2 replacements, both SCD and WC1 obtain a higher performance degradation (8% and 4%). SCD is affected more than WC1 because some of those benchmarks have a high directory occupancy with a high sharing degree (e.g., Canneal and Raytrace), and in these cases SCD needs to use more than one entry for many addresses which increases the number of directory replacements.

4.0.0.1 **Varying private cache size and core count**: Scaling private data cache size (L2 in our case) has direct impact on the number of entries that are active in the directory cache. Assuming that 100% coverage is maintained in all cases, we observe that at small private data cache sizes, single-sharer entries dominate. In this case, L2 cache replacements are frequent, which avoids exposing sharing patterns on-chip, and most addresses would be true or temporally private [12]. In such scenarios, shared addresses are rare and WC1 would approach very closely the behavior of BV. As the L2 cache size increases, so it does sharing (i.e. temporary private addresses turn into shared ones [12]), and therefore, opportunities for combining entries also grow because fewer directory entries are needed to track all the addresses stored at the L2 caches (i.e., in the case of a shared address, one directory entry tracks several entries in the L2 caches, leaving other directory entries unused due to the 100% coverage). Moreover, as most shared addresses require only a few pointers to cover all active sharers, WC1 can track them precisely by combining a few entries. For widely shared addresses (which are very few and whose number does not increase with private cache size scaling [12]) WC1 would use the coarse vector representation with one or several ways (depending on set occupation). Note that loss of precision is not so critical for widely shared lines.

Core count scaling has also impact on the number of directory entries that are active in a particular moment. In this case, however, the impact is more limited as increasing core count tends to augment sharers only for widely shared addresses [12]. When the core count is large, WC1 tracks widely shared addresses using the coarse vector representation because the associativity is never going to be large enough to have one pointer for each sharer. This way, going through larger core counts would entail minimal additional precision losses. On the other hand, for configurations with a small number of cores, the impact that precision loss has on

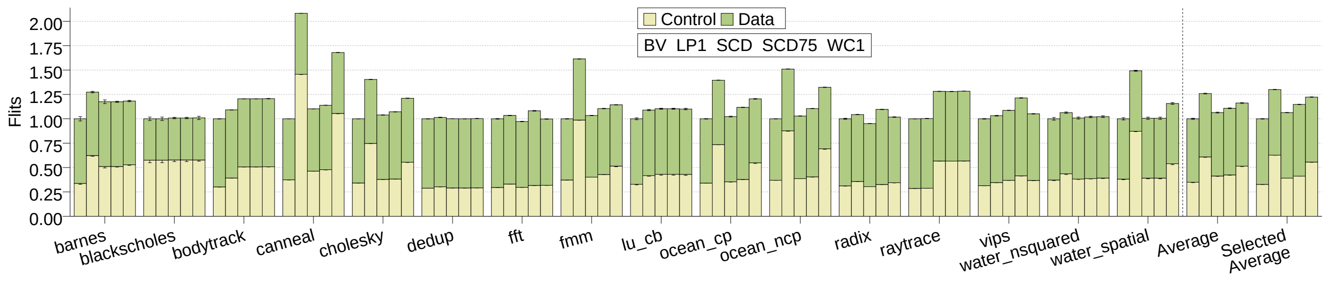


Fig. 13: Normalized total network traffic.

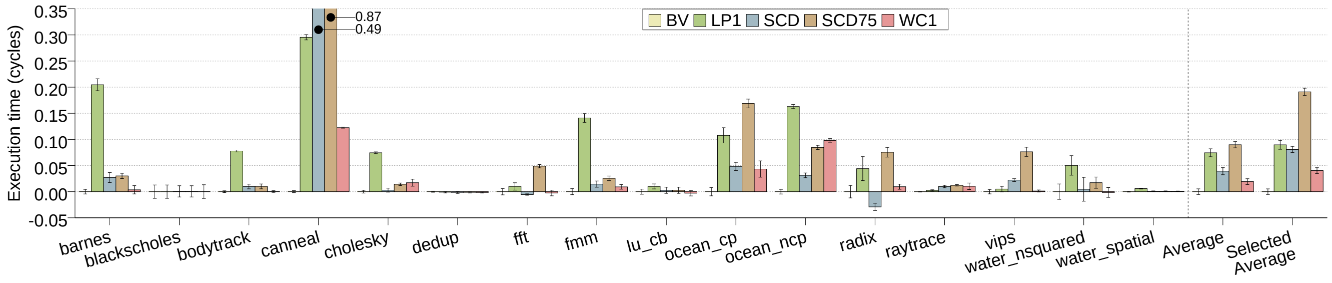


Fig. 14: Increase in the normalized execution time with respect to BV.

performance is significantly lower, and therefore, the advantage of WC1 with respect to LP1 also becomes smaller.

5 Related Work

The most common way of encoding the set of sharers of a memory block is a bit vector where each bit represents a core's local cache [25]. Unfortunately, the memory requirements of this exact and simple design grows linearly with the number of cores and thus is not scalable. The width of a directory can be reduced by codifying the sharers in an inexact way by excess, which will still guarantee correct operation of the coherence protocol. The downside of these compression techniques is that they trade off entry size for coherence traffic. Maybe the best-known example of a compression scheme is *Coarse Vector* [6].

An alternative way to reduce the width of the directory is by limiting the number of sharers that can be stored exactly in an entry. In the *Limited Pointer* scheme [14] each entry can hold a small number of pointers to sharers, which is enough for most addresses. When a memory block requires more sharers than the limit, there are two options: evicting one of the previous sharers (creating directory-induced invalidations)–Dir_iNB–or switching to an inexact representation (creating additional traffic) like using a bit to indicate that broadcast should be used to invalidate that memory block (Dir_iB) or a coarse vector that fits in place of the pointers (Dir_iCV) [6]. The number of bits required by these techniques is $i \times (1 + \lceil log_2 n \rceil)$, being i the number of stored pointers. One extra bit is required in the case of using the broadcast approach.

Simoni and Horowitz [26] enhance the limited pointers scheme by having a pool of pointers to allocate the sharers. Each entry in the pool consists of a valid bit, the identifier of node ($\lceil log_2 n \rceil$ bits), and a pointer to the next entry in the pool ($log_2 p$ bits, where p is the number of entries in the pool). Every memory block keeps a dirty bit, an empty bit, and pointer to the first sharer in the pool ($2 + log_2 p$ bits in total). Pointers are allocated in the pool on demand and, when the pool is full, evictions are performed causing invalidations. A main disadvantage of this approach is that getting the sharing information requires s sequential accesses to the pool, being s the number of sharers.

The segment directory [15] is a hybrid of the bit vector and limited pointers schemes. Each entry consist in a segment vector and a segment pointer. The segment vector is a K-bit segment of a full bit vector whereas the segment pointer is the $\lceil log_2 \frac{N}{K} \rceil$-bit field keeping the position of the segment vector within the full bit vector. The problem of this representation is that it does not adapt to the variable sharing degrees of memory blocks. Shukla and Chaudhuri employ this representation in a pool directory [17]. Also, in [27] the authors propose to design each set of a 8-way sparse directory to have six pointer ways (used to track private data) and two bit-vector ways (for keeping track of blocks with more than 1 sharer). Ways in each set are assigned to every memory block depending on its current number of sharers. All ways in WC are the same, and adaptation to varying sharing degrees is achieved by combining entries in the same set. Moreover, conversely to these proposals, WC does not rely on non-scalable bit-vectors.

In SCD [9] entries store only a limited number of pointers but they can be combined to provide more space for storing a larger number of sharers using bit vectors (hierarchically). However, to be able to do this SCD increases the size of the tags, requires the use of a Z-cache [28] and needs several directory accesses to retrieve the set of sharers. Additionally, for overflowed entries indexes to other entries must be stored, leading to reduced effective capacity of the directory. Despite these downsides, we think that SCD represents the most scalable directory coherence design to date and we have chosen it as the reference against which WC-dir is compared.

Hierarchical directories have also been proposed to reduce the entry size [29] or to navigate more efficiently the cache hierarchy [30]. However, hierarchical organizations impose additional network hops and lookups on the critical path [29] or require important modifications to the cache coherence protocol [30].

The Tagless Coherence Directory [31] uses multiple-hash bloom filter to store directory information, working similarly to an inexact duplicate-tag directory. Ideally, Tagless has constant per-core overhead, but in practice the bloom filter size needs to grow with the number of cores to avoid excessive aliasing.

Two-level directory architectures have also been proposed as a scalable way of organizing the coherence directory [32]. In a two-level directory, the first level stores the exact sharers set as a vector of bits, while the second level uses a compressed code. However, when using compression, area is saved at the expense of using an inexact representation of the sharer vector in some cases, thus yielding performance losses. In Stash [10] the second level directory information is stored along with the shared data cache and it keeps only a single bit to encode whether any core has the block. This way, entries in the first level directory are saved for private blocks.

Coherence Deactivation stores information in the directory only for shared blocks that are not read-only [8]. The rest of blocks are tracked by the page table, which acts as a second level directory at page granularity. Since most of the blocks usually tracked by the directory are private, its size can be considerably reduced. However, this proposal relies on the operating system to keep updated the non-tracked information.

Some other proposals try to exploit the fact that applications typically exhibit a limited number of sharing patterns, by storing a limited number of patterns with full bit-vectors or bloom filters in a sharing pattern table and an address-indexed sparse directory holds pointers to the pattern table [33] [34]. Although these schemes increase the range of sharers that can be tracked efficiently, they are still not scalable and require additional bandwidth.

Spatiotemporal Coherence Tracking [35] saves directory space by tracking temporarily private data in a coarse-grain fashion. Multi-grain directories [36] also uses different entry formats of the same length and tracks coherence at multiple different granularities in order to achieve scalability. However, these proposals are limited to a range of directory interleavings (those higher or equal to the size of a memory region) in order to achieve maximum benefits.

6 Conclusions

This work proposes *WC-dir*, a novel sparse directory architecture designed putting the focus on the common case, where just one pointer per entry provides enough space for tracking sharers. This way, WC-dir fits perfectly to the necessities of sequential workloads. For parallel workloads, where one pointer is not enough, our proposal takes advantage of the until now unexploited observation that several entries remain free in most sets of the sparse directory in these cases, and applies the new *way combining* concept to provide more space for sharing information to the few addresses in the set that need it. Thus, the way combining concept allows to see each set of the sparse directory as a pool of entries which are allocated dynamically as needed among the addresses mapping to that set, minimizing the storage overheads without losing the flexibility to adapt to several sharing degrees.

WC-dir can be derived with minimal changes from a sparse directory that uses the well-known Dir_1CV sharing code [6]. Like other contemporary proposals such as SCD, it can track the list of sharers through multiple formats, going from the limited pointers representation to the coarse vector one when there are no free entries left in a particular set and a new sharer needs to be added to any of the addresses in that set. However, and contrarily to SCD, WC-dir achieves this flexibility without the extra complexity of a z-cache that SCD uses, avoiding also the iterative re-insertions that keep the directory controller busy for longer times. Moreover, the fact that WC-dir remains very similar to a traditional sparse directory allows using simple replacement algorithms and simplifies directory operations.

Through detailed simulations of a 128-core architecture using a set of benchmarks exhibiting varying sharing patterns, we have shown that WC-dir reduces average execution times when compared with SCD and can practically meet the performance obtained by a non-scalable bit-vector sparse directory (just 2% overhead on average is observed). Moreover, concerning the area overhead, we have shown that for WC-dir, overhead with respect to the private caches is lower than SCD's for 128 cores, and moreover it remains constant as we increase the number of cores, whereas SCD grows albeit slowly. The only downside that we have observed for WC-dir is some more extra network traffic. Particularly, WC-dir increases traffic 6% on average when compared with a similarly sized SCD (SCD75 configuration) and 10% compared with a SCD configuration with the same number of entries, which requires 25% more area. Observe, however, that the WC1 design evaluated in this work puts the emphasis on minimizing area overhead while maintaining the execution time. The area requirements can be increased in exchange of reduced traffic by, for example, duplicating the number of bits per entry (and thus the number of initial pointers and the size of the coarse vectors) in WC-dir would cut down the traffic penalty whilst still preserving advantages over SCD (lower execution time, less area —although to a lesser extent— and simpler implementation).

Acknowledgments

This work has been supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant "TIN2015-66972-C5-3-R".

References

[1] M. M. K. 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, Jul. 2012.

[2] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture, M. D. Hill, Ed. Morgan & Claypool Publishers, 2011.

[3] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016.

[4] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array," in *2016 Symposium on VLSI Technology and Circuits*, Jun. 2016, pp. 1–2.

[5] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A scalable architecture based on single-chip multiprocessing," in *27th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2000, pp. 12–14.

[6] A. Gupta, W.-D. Weber, and T. C. Mowry, "Reducing memory traffic requirements for scalable directory-based cache coherence schemes," in *19th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1990, pp. 312–321.

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

[8] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2011, pp. 93–103.

[9] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *18th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2012, pp. 129–140.

[10] S. Demetriades and S. Cho, "Stash directory: A scalable directory for many-core coherence," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 177–188.

[11] L. Zhang, D. B. Strukov, H. Saadeldeen, D. Fan, M. Zhang, and D. Franklin, "Spongedirectory: Flexible sparse directories utilizing multi-level memristors," in *23rd Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2014, pp. 61–74.

[12] M. Zhao and D. Yeung, "Studying the impact of multicore processor scaling on directory techniques via reuse distance analysis," in *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 590–602.

[13] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[14] A. Agarwal, R. Simoni, J. L. Hennessy, and M. A. Horowitz, "An evaluation of directory schemes for cache coherence," in *15th Int'l Symp. on Computer Architecture (ISCA)*, May 1988, pp. 280–289.

[15] J. H. Choi and K. H. Park, "Segment directory enhancing the limited directory cache coherence schemes," in *13th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr. 1999, pp. 258–267.

[16] S. S. Mukherjee and M. D. Hill, "An evaluation of directory protocols for medium-scale shared-memory multiprocessors," in *8th Int'l Conf. on Supercomputing (ICS)*, Jul. 1994, pp. 64–74.

[17] S. Shukla and M. Chaudhuri, "Pool directory: Efficient coherence tracking with dynamic direcory allocation in many-core systems," in *33rd Int'l Conf. on Computer Design (ICCD)*, Oct. 2015, pp. 557–564.

[18] 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," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[20] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *Computer Architecture News*, vol. 37, no. 2, pp. 10–19, Jul. 2009.

[21] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[22] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Jan. 2011.

[23] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[24] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 196–207.

[25] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers (TC)*, vol. 27, no. 12, pp. 1112–1118, Dec. 1978.

[26] R. Simoni and M. A. Horowitz, "Dynamic pointer allocation for scalable cache coherence directories," in *Int'l Symp. on Shared Memory Multiprocessing*, Apr. 1991, pp. 72–81.

[27] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang, "Building expressive, area-efficient coherence directories," in *22nd Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2013, pp. 299–308.

[28] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2010, pp. 187–198.

[29] S.-L. Guo, H.-X. Wang, Y.-B. Xue, C.-M. Li, and D.-S. Wang, "Hierarchical cache directory for cmp," *Journal of Computer Science and Technology*, vol. 25, no. 2, pp. 246–256, Mar. 2010.

[30] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "A split cache hierarchy for enabling data-oriented optimizations," in *23th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2017.

[31] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A tagless coherence directory," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 423–434.

[32] M. E. Acacio, J. González, J. M. García, and J. Duato, "A new scalable directory architecture for large-scale multiprocessors," in *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 97–106.

[33] H. Zhao, A. Shriraman, and S. Dwarkadas, "SPACE: Sharing pattern-based directory coherence for multicore scalability," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 135–146.

[34] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan, "SPATL: Honey, i shrunk the coherence directory," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2011, pp. 148–157.

[35] M. Alisafaee, "Spatiotemporal coherence tracking," in *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2012, pp. 341–350.

[36] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 359–370.