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—This manuscript opens the way to a new class of coherence directory structures that are based on the brand-new concept of *way combining*. A Way-Combining Directory (WC-dir) builds on a typical sparse directory but allows to take advantage of several ways in the same set to codify the sharing information of each memory block. The result is a sparse directory with variable effective associativity per set and variable length entries, thus being able to dynamically adapt the directory structure to the particular requirements of each application. In particular, our proposal uses just enough bits per entry to store a single pointer, which is optimal for the common case of having just one sharer. For those addresses that have more than one sharer, we have observed that in the majority of cases *extra* bits could be taken from other empty ways in the same set. All in all, 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. Detailed simulations of a 128-core multicore architecture running benchmarks from PARSEC-3.0 and SPLASH-3 demonstrate that WC-dir can closely approach the performance of a non-scalable bit vector sparse directory, beating the state-of-the-art Scalable Coherence Directory (SCD) and Pool directory 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 enable 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], and large-scale NoCs for supporting them [5]. 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 [6] are plainly and simply unfeasible for manycores, since their associativity grows with the number of cores. Contrarily, sparse directories [7] 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 [8], [9], [10], [11], [12], [13].

Two aspects determine the area requirements of a sparse directory [14]: 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 [10]) 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 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) [10]. 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 [15], [16] or compressed sharing codes [7], [17] 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

- [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

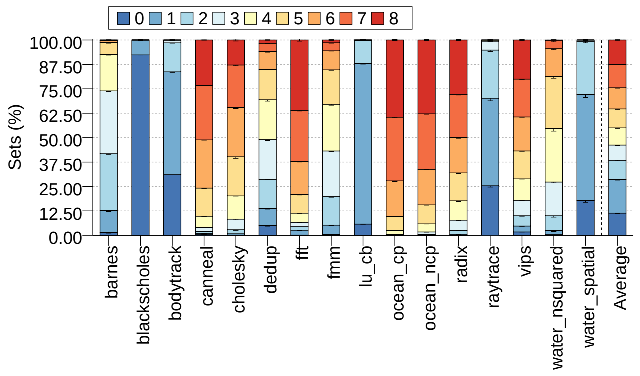


Fig. 1: Directory occupancy per set: average fraction of sets with a given number of occupied entries (ways) in a 100% coverage 8-way sparse directory with bit vector sharing code for 128 cores (1 sample every 100000 cycles).

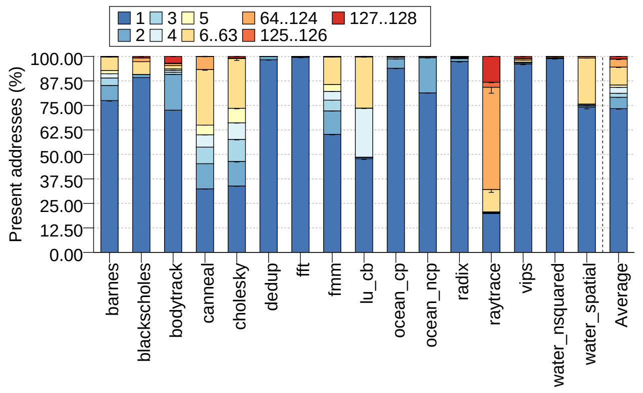


Fig. 2: Sharers per directory entry: average fraction of present addresses with a given number of sharers in a 100% coverage 8-way sparse directory with bit vector sharing code for 128 cores (1 sample every 100000 cycles).

levels. For instance, the loss of precision introduced by coarse bit vectors [7] leads to more invalidation messages per write, while pointer recycling policies [15] must invalidate privately cached blocks every time a pointer is reused for a new sharer. In 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 [9], [18]. 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 [13]. 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 blocks being shared by several cores and occupying a single directory entry. For the benchmarks considered in this work, Fig. 1 shows that sets are on average at half their maximum occupation, and Fig. 2 depicts the number of sharers tracked by each entry (refer to Section 4 for details). Interestingly, most of those applications that exhibit high occupancy in Fig. 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 [10], [19]. 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 combination*. 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 combination comes with minimal cost as it avoids the complexity introduced by other proposals [8], [10], [19]. Our proposal builds atop traditional sparse directories, relies on existing replacement algorithms, and does not increase the complexity of directory operations. Although our proposal is less flexible than SCD [10], 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 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 it improves over a similar-size version of the previously proposed SCD directory and the bigger Pool directory.

This manuscript extends our previous conference paper [20] by presenting a more mature (more finely-tuned) proposal for a *way-combining* directory, showing its low hardware complexity and significantly extending the evaluation of the idea. In particular, we make the following contributions in this extended version:

- We describe in detail how read and update operations would proceed in WC-dir, showing that our proposal would introduce no penalty in terms of extra latency and addressing some implementation details: policies for replacements, possibility of hybrid representations and allowed sizes for the coarse bit vector representation (Section 3).

- We add a new comparison point to the evaluation. In particular, besides SCD [10], we also consider the Pool directory [19], as it can scale gracefully and was previously reported to be a competitive alternative to non-scalable bit vector directories (Section 5).
- We make a more comprehensive evaluation by presenting some new results that help understand how WC-dir is able to beat its counterparts: obtained precision per address, entries used in each format per address, directory replacements per instruction (Section 5).
- We also add a sensitivity analysis to illustrate the behavior of WC-dir with varying directory associativity, showing how the lack of precision in the directory can be attributed to the lack of directory associativity (Section 5).

The rest of the manuscript is organized as follows. First, we give in Section 2 background information about directory cache coherence and discuss some important related works. Next, we present our proposed directory architecture in Section 3. Section 4 describes our simulation environment and detailed results are shown and analyzed in Section 5. Finally, Section 6 contains the main conclusions of this work.

2 Background and 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 [21]. 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* [7].

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 [15] 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) [7]. 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 [22] 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 node identifier ($\lceil log_2 n \rceil$ 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 [16] is a hybrid of the bit vector and limited pointers schemes. Each entry consists of 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. Also, in [23] the authors propose to design each set of an 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. Recently, Shukla and Chaudhuri employ a segment directory representation in combination with limited pointers in a pool directory [19].

In SCD [10] 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 [24] 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 [25] or to navigate more efficiently the cache hierarchy [26]. However, hierarchical organizations impose additional network hops and lookups on the critical path [25] or require important modifications to the cache coherence protocol [26].

The Tagless Coherence Directory [27] 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 [28]. 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 [11] 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 [9]. 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

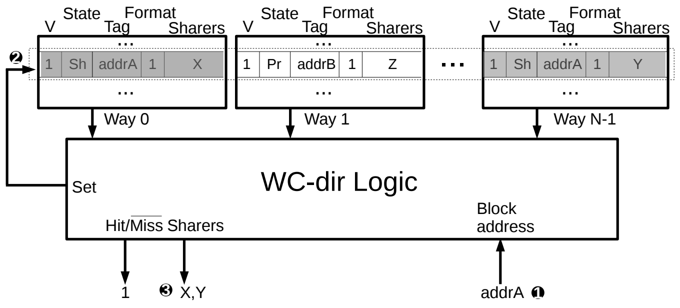


Fig. 3: Overview of the Way-Combining Directory. The block address `AddrA` is presented to the directory ❶. The corresponding set is accessed and two of the ways (`Way 0` and `Way N-1`) happen to store the sharing information for the address ❷. From the contents of these two ways, the list of the sharers (`X, Y`) is generated ❸.

sharing pattern table and an address-indexed sparse directory holds pointers to the pattern table [29] [30]. 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 [31] saves directory space by tracking temporarily private data in a coarse-grain fashion. Multi-grain directories [32] also use 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.

3 The Way-Combining Directory

3.1 General Overview

Like any other coherence directory, 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 manycore chip multiprocessor.

The structure of WC-dir is nearly identical to that of a traditional set-associative directory cache. 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. Fig. 3 gives a simple overview of a circuit for obtaining the list of sharers. 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 the *WC-dir Logic*, a finite state machine (FSM) whose purpose is to merge the sharing information from all the matching entries (see Section 3.3 for further details).

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 multiprogrammed workloads. Nevertheless, even 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, every additional sharer for those addresses already tracked by the directory will result in an additional free directory entry. 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 [7] (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 since more nodes than real sharers are typically included). 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 0, none of them hold a copy.

The list of sharers is jointly stored by all combined entries and can be decoded using the referred WC-dir 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 at a given time.

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 as long as there are free entries in the same set of the directory. However, when directory resources become insufficient to maintain exact sharing information, the format of the stored information of some addresses changes to coarse bit vector format (losing some precision) or its amount of storage is dynamically reduced (losing more precision).

This way, WC-dir dynamically changes the amount of sharing code storage dedicated to each address in an attempt to maximize directory utilization and precision while keeping low area overhead and operation complexity.

When adding a new address to the directory in a set that is full (i.e., a set where all the entries are valid) but contains at least one combined entry, WC-dir will reduce the storage of a combined entry, hence reducing precision. If an address in a combined entry in coarse format exists, WC-dir makes room by decreasing its number of allocated entries. Otherwise, WC-dir entries allocated to an address in pointer format are switched to coarse format using at least one fewer entry.

Since evicting an address from the directory results in invalidations in private caches that may later harm performance due to additional misses, WC-dir always tries to minimize evictions at the cost of reducing the precision of the sharing code. Thus,

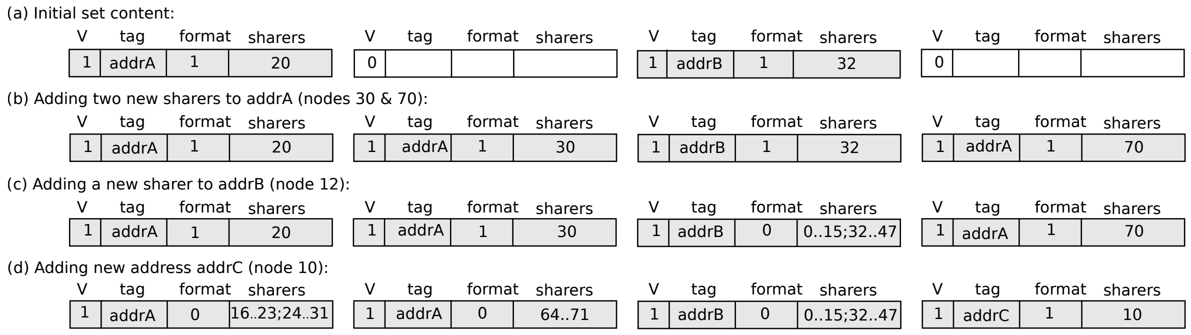


Fig. 4: WC-dir: Example of operation.

evictions only occur when a new address is inserted into a full set where each entry is allocated to a different address, following a modified LRU replacement algorithm to select the victim. When adding a sharer to an existing address, no eviction or reduction in precision of the sharing information of another address happens.

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

3.2 Working Example

To illustrate the behavioral aspects of WC-dir, Fig. 4 shows the evolution of a 4-way set associative WC-dir for 128 nodes. Each sharer field consists of 8 bits that, in pointer format, can be combined to point up to 4 sharers (one per cache way) or, in coarse format, to compose a 32-bit (4×8) or 16-bit (2×8) sharer vector, each bit representing 4 ($128/32$) or 8 ($128/16$) nodes respectively.

Fig. 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 Fig. 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, as shown in Fig. 4 (c). In this case, *addrB* must change its representation from pointer format to coarse bit vector format, therefore losing precision (note that a total of 32 potential sharers would be actually encoded).

Finally, the insertion of a new address (*addrC*) in the set (Fig. 4 (d)), causes *addrA* to switch from pointer format over three ways to coarse vector format over two ways, thus releasing one of its entries and losing some precision (note that no address is evicted). The replacement algorithm implemented in WC-dir starts looking for the candidate among those addresses that comprise several ways in the set, and among these, those in coarse bit vector format are considered in the first place. Though in this example there is only one candidate, in practice there are several heuristics that could be employed to select the victim among the candidate addresses. In this work, WC-dir opts for a simple LRU policy, although other approaches could be used (further details are given in the next subsection). 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.

3.3 Implementation aspects

The proposed architecture can be implemented using simple hardware with negligible or no delay over conventional directory approaches. This section illustrates this claim by discussing, from a high-level perspective, implementation issues required to support the two major directory operations, reading and updating contents.

WC-dir builds on a typical set-associative directory cache structure. Figure 5 presents a block diagram of the WC-dir directory, where the contents of each cache way (e.g. tag and sharers) are connected to the *WC-dir logic*, which is the FSM responsible for carrying out both update and read operations of coherence information. The *coherence controller* is in charge of receiving the requests (generated on cache misses and replacements) and coherence responses from the last-level private caches, and creating and injecting the corresponding response and coherence messages into the NoC. As in a typical sparse directory, the coherence controller in WC-dir has a buffer (the miss status handling registers, MSHRs) that holds the requests received from the L2 caches on cache misses and replacements until they are completely resolved. On receiving a new request, the buffer is checked to look for a potential in-progress request to the same block address. If one is found, a pending bit in the entry allocated to the new request is set to indicate that it must wait for a previous request to the same address to be completed. Otherwise, the request may be processed once the FSM implemented by the coherence controller chooses it. In our implementation, the coherence controller handles one request in each cycle (coherence controller cycle), and therefore, a single access is sent to the WC-dir Logic every time. If the request to the WC-dir Logic takes more than one cycle to complete (e.g., in a *GetX* request, for getting the sharers list from a combined entry), the coherence logic will not be able to proceed to handle another request until it has received the complete response. Observe, however, that several requests to different addresses will typically be in-progress concurrently (for example, the coherence controller may start processing a different request while waiting for some in-flight coherence responses).

The coherence controller indicates to the WC-dir logic both the type of the access (*GetS* for requesting shared data, *GetX* for requesting exclusive data, *PutS* due to the eviction of shared data or *PutX* due to the eviction of exclusive data) and the block address involved through the $\mathtt{I}_{ReqType}$ and $\mathtt{I}_{Address}$ entries respectively. Then, the WC-dir logic performs a read operation to obtain coherence information for the block address (the set indicated by $\mathtt{O}_{SetSelect}$), and informs the coherence logic on whether there is a hit or a miss ($\mathtt{O}_{Hit/Miss}$), and in case of a directory hit, the coherence state of the memory block ($\mathtt{O}_{State}$) and sharing information ($\mathtt{O}_{Sharers}$). The latter includes the format being used (i.e. pointer or coarse bit vector), the number of chunks that sharing information is split in (i.e. number of ways used to store

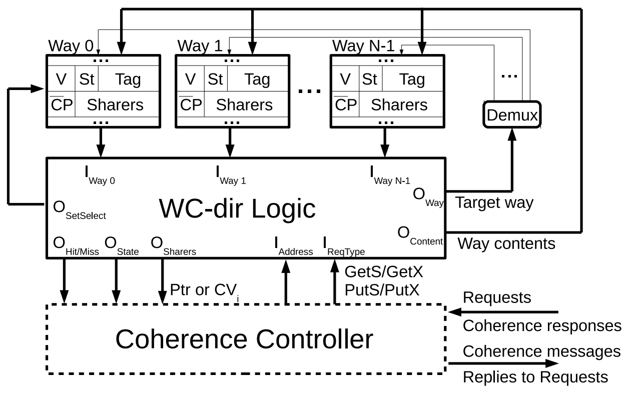


Fig. 5: WC-dir proposed implementation.

sharing information) and the contents of a particular way (as we explain later, the list of pointers or the full coarse bit vector will be provided using several directory cycles). Finally, directory information is updated by re-writing one by one the involved ways in the corresponding set (using O_{Way} and $O_{Content}$ outputs).

The implementation aspects of the read and update operations are drawn in Fig. 5.

3.3.1 Reading directory information

On receiving a GetS or GetX request, the contents of a target directory entry must be read to find out the owner of the missing block or the list of sharers for coherence purposes (such as forwarding the request to the private cache currently owing the memory block or invalidating the shared copies of the block in the private caches). Unlike update operations, directory read operations are on the critical path of cache misses, so it is important that they take as little time as possible. In this context, the proposed WC-dir design is aimed at avoiding extra delays in directory read operations when compared to a traditional directory.

The time taken to read the contents of the target directory entry mainly depends on whether or not it is a combined entry (i.e. whether it comprises just one or several ways in the set). Reading a non-combined entry, regardless of whether the entry uses pointer format or coarse bit vector format, is performed identically as in a traditional sparse directory, since our approach only impacts on combined entries. Thus, there is not additional penalty over the baseline in this case. As it will be shown in Subsection 5.3, this is the common case for most applications, constituting about 85% of the cases on average.

However, the proposal also needs to take care of combined entries, especially when a small number of ways in the set are involved. To this end, WC-dir proceeds as follows on accessing directory information. First, the tag of the target block address is compared against the tags of all the ways in the set and the sharers, format and state fields are read, as usual. The tag comparison produces a *bit mask* with the combined ways of the target entry (e.g., 1010, where 1 refers to a way with a matching tag and 0 refers to a miss, so there are combined entries if there is more than one 1). Second, the bit mask is used to discard the contents (sharers, format and state fields) of the non matching ways in the set. Third, the remaining contents are sent to the coherence controller one by one, which uses them to identify the private caches involved in the coherence operation.

For instance, if the combined entry is encoded in two ways in the set with pointers P1 and P2, then these pointers have to be delivered by the WC-dir logic to the coherence controller, as depicted in the bottom side of Fig. 5 along with the coherence state of the memory block and the format being used in the codification (pointer format in this case). Since multicast support is not commonly employed in current on-chip networks, the coherence controller must create individual coherence messages sequentially (one by one), thus requiring several directory cycles. This way, while the coherence controller is creating one coherence message, the WC-dir logic is providing it with the next sharer or set of sharers, thus overlapping the time needed to obtain additional contents in the case of a combined entry with the time to create coherence messages.

In case the combined entry is in coarse bit vector format, the chunks of the full coarse bit vector would also be provided to the coherence controller one by one. The number of chunks is also sent since the beginning, which allows the coherence controller to determine how many sharers are represented by each bit. This way, the coherence controller can start generating the coherence messages upon receiving the first chunk, overlapping again the time needed to provide the additional chunks with the time needed to create coherence messages.

To keep hardware as simple as possible, we enforce that the number of ways in the set of a combined entry in coarse bit vector format is constrained to be a power of two (e.g. 1, 2, 4 or 8 in an 8-way cache design). This allows the coherence controller to employ the same logic used in a standard directory (like LP1 as described in section 4.1) to encode or decode the coarse bit vector with very simple modifications (i.e., adding a multiplexer and replicating wires). Through experimentation, we have found out that the impact that this restriction has on precision or performance is negligible.

PutS and PutX commands would require reading the directory to obtain the information for the involved block address prior to update. In these cases, the same course of action would be followed.

3.3.2 Updating coherence information

Directory information must be updated as a consequence of the requests (GetS or GetX) generated on cache misses, or the associated replacement notifications (PutS or PutX). For the latter, updates on receiving a PutX command (that informs about replacement of an exclusively held memory block) would proceed exactly the same way as in a traditional sparse directory by releasing the single-way directory entry associated to the block address. Otherwise, in those protocols that implement noisy replacements of shared data (something beneficial for directory structures with limited-capacity entries such as SCD or WC-dir), only when directory information is stored in pointer format, can the portion of the entry tracking the replaced copy of the memory block be released. Notice that if, otherwise, the coarse bit vector representation is used, the replacement hint cannot be leveraged as a consequence of the lack of precision entailed by this sharing code. Overall, replacements do not suppose any changes regarding a traditional sparse directory.

Regarding requests (GetS or GetX) generated on cache misses, if the directory already maintains an entry for the requested address (directory hit), the list of sharers must be updated. Otherwise, a new directory entry needs to be created. In the latter case, as a result of the limited capacity of the directory structure, another tracked address could also require updating its associated entry.

TABLE 1: WC-dir update logic in case of address hit.

GetX		
Pointer	**Coarse**	**Action**: *establish requester as new owner*
•		Use one entry in pointer format, release the rest
	•	Use one entry in pointer format, release the rest
GetS		
Pointer / Free way	**Coarse**	**Action**: *a new sharer is to be added*
✓		Use a free entry in pointer format
✗		Combine pointers in coarse vector format
	•	Update the coarse vector to include the new sharer

TABLE 2: WC-dir update logic in case of address miss.

GetX / GetS			
Free way	**Combined entries**		**Action**: *new entry and shared added*
	Coarse	Pointer	
✓			Use a free entry in pointer format
✗	✓		Release one or more ways by re-encoding the LRU coarse vector entry
✗	✗	✓	Release one or more ways by re-formating the LRU pointer entry into coarse vector
✗	✗	✗	Release one way by by evicting the LRU entry

Below, we first discuss the actions performed in case of a directory hit, and next those involved in a directory miss.

Upon a directory hit for a GetX request, the directory entry must be updated to reflect the identity of the new owner of the block. In this case, a single pointer suffices, so that if the directory entry is comprised of several combined ways, only one of them is kept, deallocating the rest and leaving them available for later use.

On the other hand, if the directory hit happens for a GetS request, the directory entry must be updated so as to add the new sharer that triggered the request. Now, the actions to be done depend on both the format being used to track that address and whether or not there is any free way in the corresponding set of the directory cache. In case the sharers are being tracked in pointer format and there is at least a free way in the set, then a way is taken and the new sharer encoded in pointer format; otherwise, if all the ways are already in use, the representation is changed and the pointers of the tracked block are combined in the coarse bit vector format and the new sharer subsequently added to the vector.

On the other hand, if the sharers are being tracked in coarse bit vector format then the new sharer must be included among the existing ones. In this case, it does not matter whether or not there is any available free way in the set, because the representation maintains its current setup (number of combined ways and format). Notice that finding out available ways in the set in this case would open the door to hybrid-codification directory entries, in which part of the sharers are codified in coarse bit vector format and others using pointers (in this case, the newly added sharer). Having this hybrid representation however would complicate the hardware implementation, and through experimentation, we have observed that it would lead to negligible improvements in the precision of the directory. Therefore, we do not consider it and we force that all the ways tracking a given memory block must use the same format.

Table 1 summarizes the main actions that must be taken by the WC-dir controller in case of an address hit in the directory cache.

To accomplish these updates, the *WC-dir logic* would select the target ways to be updated and would generate the new content (i.e. pointers or chunks of the coarse bit vector) of each way. Through a demultiplexer, the target way (of the corresponding set) would be selected and subsequently updated with the contents generated by the WC-dir logic, as depicted in Fig. 5. Notice that only one way can be updated in every directory cycle, hence, the total number of directory cycles needed to complete an update operation depends on the number of involved ways. However, as directory updates are not on the critical path of misses, these additional cycles would have negligible impact on performance.

On the other hand, the way to proceed upon a directory miss is the same for GetS and GetX requests. In both cases, the directory must be updated to track the missing memory block address. For this purpose, firstly, it is checked whether or not there is any free cache way in the target directory cache set. In such a case, the free way is allocated and a new directory entry is created to track the missing block address in pointer format.

Otherwise, room must be made by releasing one of the currently used ways in the set and therefore the contents of two addresses should be updated in the directory structure (the address being inserted and the address providing the resources that it needs). For this purpose, the approach prioritizes combined entries over non combined ones, so as to maximize the number of addresses that are tracked (and thus to minimize private cache misses). Among the combined entries, those in coarse bit vector format are prioritized over those in pointer format, so as to try to cause minimal impact on precision. In other words, firstly, if there are several combined entries in coarse bit vector format, the LRU one is selected, and the size of the coarse bit vector is reduced so as to release a way that will be used to track the address being accessed. Otherwise, the LRU combined entry in pointer format would be selected, and its information would be recoded in coarse vector format using one fewer entry, thus losing precision. We have also considered and evaluated other replacement alternatives, obtaining the same or slightly worse results for them. Overall, we have observed that the election of the heuristics used to reclaim space for an incoming new address has low impact on performance. For this reason, we opt for the alternative that mimics more closely what would happen in the baseline.

Remember that to keep hardware as simple as possible, we assume that the number of ways of a combined entry in coarse vector format is constrained to be a power of two (e.g., 1, 2, 4 or 8 in an 8-way cache design). Therefore, more than one way may be released as a result of compacting a combined entry both in coarse vector format or in pointer format.

Finally, if there are not any combined entries in the set, then the LRU entry is evicted as would also occur in a traditional sparse directory. Table 2 summarizes the discussed actions. The procedure followed for updating directory information would be the same as that explained above for directory hits.

4 Evaluation Methodology

We evaluate the performance of different cache coherence directories using the GEMS 2.1 simulator [33]. GEMS is fed with information gathered by a PIN tool [34], which offers detailed information about the instructions executed, memory references, and synchronization primitives as is the standard methodology for large-scale system simulations [35]. We model the interconnection network with Garnet [36]. 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 3.

TABLE 3: 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)
Pool size (Pool)	512 entries, 4 pointers per entry
Directory latency	5 cycles
Physical address size	48 bits
Memory access time	200 cycles
Network parameters	
Topology and Routing	2-D mesh (16×8), X-Y
Flit size	16 bytes
Message size	5 flits (data), 1 flit (control)
Link time	2 cycles
Bandwidth	1 flit per cycle

TABLE 4: 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

Our simulations consider representative applications from PARSEC 3.0 [37] and SPLASH-3 [38] (see Table 4). We have included as many benchmarks as we have been able to. 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 simulation 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 by repeating each execution 4 times.

4.1 Evaluated directory configurations

We evaluate six configurations for the coherence directory that we name BV, LP1, SCD, SCD75, Pool, 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 [7] 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 [10] 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 our proposed WC1, since it uses a 3-way z-cache that explores four levels (roughly equivalent to a 45-way cache). Pool is an implementation of the pool directory [19] that has a 512-entry pool with 4 pointers per pool entry. Finally, WC1 is an implementation of WC-dir that uses 1-pointer entries. BV and LP1 use silent replacements of shared blocks [39] (no notification is sent to the directory in case of eviction of a clean shared block) and WC1, Pool, 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 policy for each case.

4.1.1 Memory requirements

Table 5 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 performance evaluation, memory requirements for 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. Pool requires always some more memory that WC1 or LP1, and its overheads grows with the number of cores as the pointers of the pool need more bits, but it grows slower than SCD's.

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. Pool needs the same bits for sharing information as WC1 and LP1 for 512 nodes or more, but it needs a bit more for smaller sizes due to the chosen pool size (512 entries), because the pointer stored in the directory is larger. Moreover, the memory requirements of the pool also grow logarithmically with the number of nodes, although we keep the number of entries 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.

5 Performance Evaluation

5.1 Directory performance

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, how precise the information

TABLE 5: Directory size and overhead for different configurations (LP1 sizes are identical to WC1).

Nodes	64					128					256					512					1024				
Directory	BV	SCD	SCD75	Pool	WC1	BV	SCD	SCD75	Pool	WC1	BV	SCD	SCD75	Pool	WC1	BV	SCD	SCD75	Pool	WC1	BV	SCD	SCD75	Pool	WC1
Tag (bits)	28	36	36	28	28	27	35	35	27	27	26	34	34	26	26	25	33	33	25	25	24	32	32	24	24
Sharing Code (bits)	64	11	11	10	7	128	16	16	10	8	256	20	20	10	9	512	28	28	10	10	1024	37	37	11	11
Pool / Tile (KiB)				2.4					2.7					2.9					3.2					3.4	
Size / Tile (KiB)	23.5	12.3	9.2	10.0	9.3	39.3	13.3	9.9	9.8	9.3	71.0	14.0	10.5	9.5	9.3	134.8	15.8	11.8	9.3	9.3	262.5	17.8	13.3	9.3	9.3
% over L2	17.2	8.9	6.7	9.1	6.8	28.6	9.7	7.3	9.1	6.8	51.8	10.2	7.7	9.1	6.8	98.4	11.5	8.6	9.1	6.8	191.6	13.0	9.7	9.3	6.8

is, and how much information can be simultaneusly stored. An ideal directory must have low access latency to obtain the sharing information, perfect precision, and no replacements of sharing information due to directory capacity (or conflicts).

In this first part of the evaluation, we focus on these three aspects: the precision, the number of directory replacements, and the latency at the L3 (where the directory is placed). In some cases, a directory will increase the number of tracked sharers by reducing the precision of the stored information (always by storing a superset of the actual sharer set) at the cost of more invalidation traffic. In other cases, a directory will stop tracking some sharers in benefit of new ones by evicting entries and invalidating the corresponding cached copies, causing extra cache misses.

Fig. 6 shows the average precision per address stored in the directory during the whole execution of the applications (we take a sample of all addresses in the directory every 100 000 cycles and report the overall average). As shown in Equation 1, this is measured for each address as the number of actual sharers divided by the number of sharers encoded by the sharing code, being `A` the total number of block addresses and `N` the number of samples[1].

$$Precision = \frac{1}{N}\sum^{N} \frac{1}{A}\sum^{A} \frac{\#Real\,Sharers}{\#Codified\,Sharers} \tag{1}$$

Both SCD and Pool achieve perfect precision with much fewer resources than BV. LP1 and WC1, using even fewer resources, 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 Fig. 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.

Fig. 7 plots the number of directory replacements per instruction. As already explained in Section 3, 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 (see Fig. 6). 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 4, 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. Fig. 8 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. Pool also has a lower number of L2 replacements because sometimes, in order to be able to add a new sharer to an address, it first has to make room in the pool invalidating sharers from other addresses, hence creating empty entries in the L2 caches.

Regarding the access latency to the directory information, arguably, the only two implemented directory schemes that increase access latency are SCD and Pool directory. The reason is that SCD and Pool may require several sequential accesses to gather the whole directory information. Although in most of the cases they find the directory information with only one access (e.g., private blocks), for some other cases this extra latency can noticeably affect the average L2 miss latency. Fig. 9 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). The *At_L3* time of SCD, SCD75 can increase for some benchmarks (e.g., Canneal). But more interestingly, the *At_L3* time of Pool increases for even more benchmarks and to a larger extent (e.g., Barnes, Bodytrack, Canneal, Fmm, Ocean_cp, Water_nsquared, and Water_spatial). This extra latency in the Pool directory is caused by extra directory accesses to collect the sharing information and it affects the overall miss latency (e.g., Canneal, Fmm, and Water_spatial).

Fig. 9 also shows 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

1. The actual number of sharers for each address is calculated by counting the number of L2 caches holding a copy of the memory block at each measuring point. Note that the resulting value is less than or equal to the number of sharers codified in BV, as this configuration implements silent replacements for clean shared data (a node may cease being a sharer of an address without notifying the directory). This is the reason why BV does not reach 100% precision in some cases.

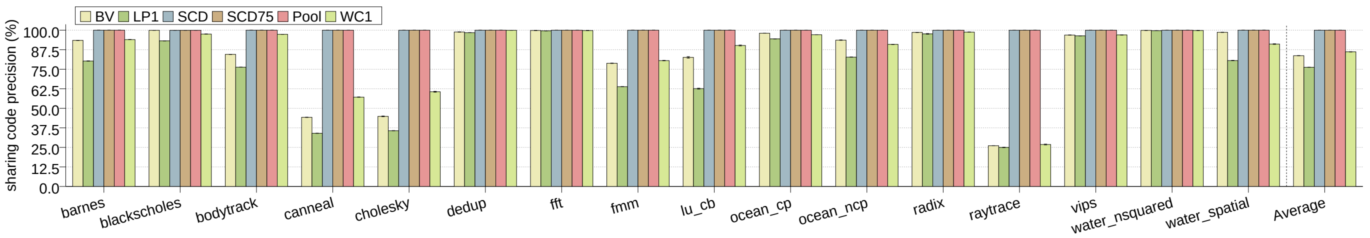


Fig. 6: Precision per address measured as the average for each address of the ratios between the actual number of sharers and the number of sharers encoded in the directory. The directory is sampled every 100 000 cycles.

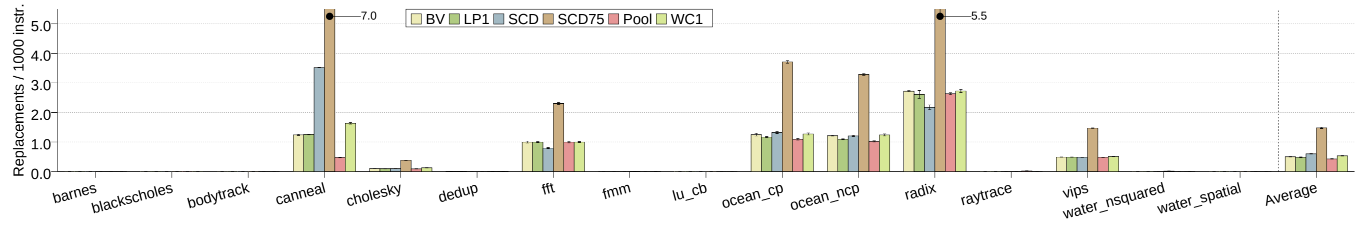


Fig. 7: Directory replacements per instruction.

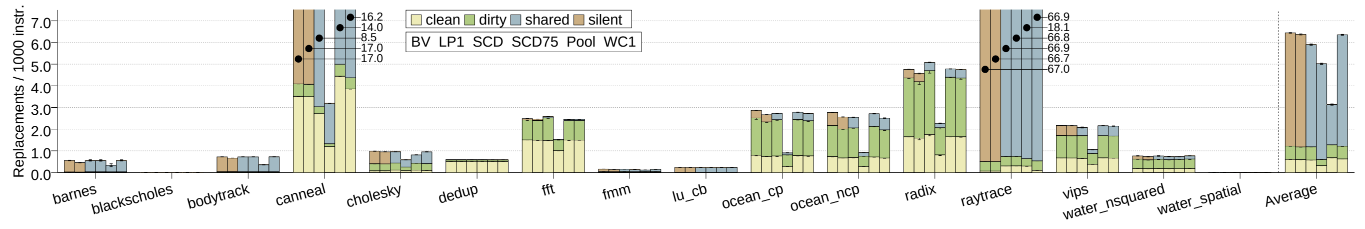


Fig. 8: L2 cache replacements per instruction.

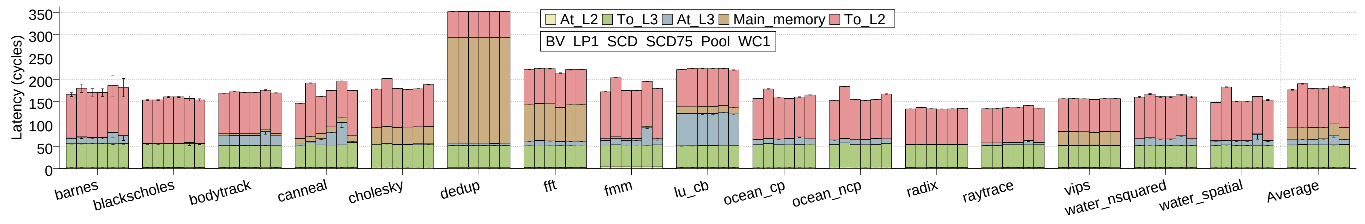


Fig. 9: Breakdown of L2 miss latency.

on write misses due to the lack of precision of their sharing information, as we will see in the next subsection. 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).

5.2 Impact on network traffic and execution time

The most direct effect of the lack of precision and capacity of the directory information is that unnecessary invalidation messages are sent upon write misses, as shown in Fig. 10, and upon directory replacements. These extra messages can have in some cases significant effect in the total network traffic, as shown in Fig. 11. Particularly, we plot in this figure the total number of flits generated along applications' execution. Note, thus, that there is no correlation among the amount of extra traffic shown in Fig. 11 and the increases in L2 average latency shown in Fig. 9. For most benchmarks, the increase in traffic does not have an important effect on miss latency, as can be seen in Fig. 9, and hence will not affect the execution time in a significant extent. Here again we see that the increased precision afforded by the way combination technique allows WC1 to have much lower traffic than LP1, although it is still higher than BV's, SCD's, and Pool's. Interestingly, though SCD and Pool reach perfect precision, the difference in average traffic regarding WC1 is just 10%, even though SCD and Pool have 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 [40]. 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

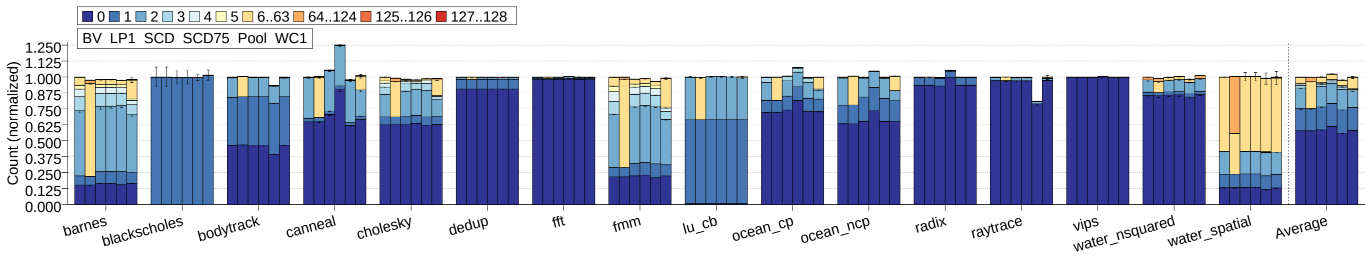


Fig. 10: Frequency of each number of sharers invalidated per L2 write miss.

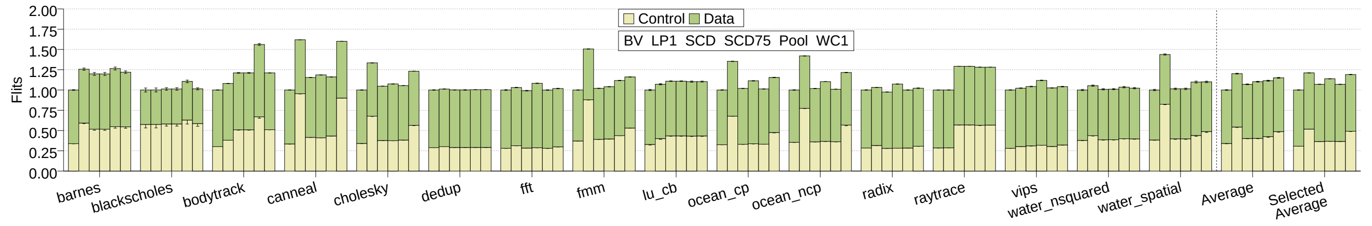


Fig. 11: Normalized total network traffic.

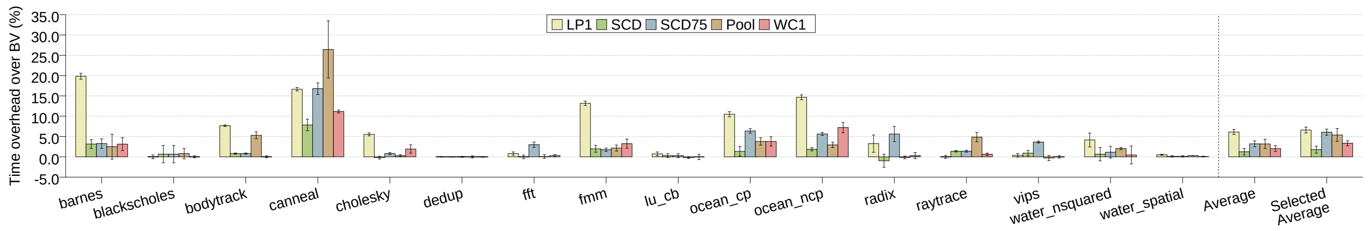


Fig. 12: Increase in the normalized execution time with respect to BV.

energy consumption is minimal.

Fig. 12 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 1.3% 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. Additionally, WC1 average overhead with respect to BV is just 2.1%. If we look only at those benchmarks with many L2 replacements, both SCD and WC1 obtain a higher performance degradation (1.8% and 3.4%). Finally, Pool entails higher performance degradation than WC1 (3%) due to the extra latency accessing the directory, for example, in Canneal.

5.3 Sensitivity analysis of associativity in WC

As already discussed, WC-dir combines entries that belong to the same directory set. Therefore, the more associativity the directory has, the more combining opportunities appear. This section analyzes the behavior of WC-dir with varying associativity and discusses the limit of these opportunities.

The directory tracks copies of blocks cached in the private caches. However, a block in a private cache cannot be placed at an arbitrary position, but it has to be placed at a particular set, commonly given by the least significant bits of the block address. On the other hand, directory information is distributed between several banks (i.e., one bank per tile in the configuration assumed in this work), and the sharing information for a memory block is also placed in a particular bank and set, commonly using again the least significant bits of the block address to select the bank, first, and set, then. If a directory with 100% coverage (same number of directory entries as last-level private cache entries) is designed with enough associativity, such that the address bits used for indexing the cache matches the bits used for indexing the directory bank and set, then directory conflicts would vanish and there would be a possible one-to-one mapping between the cache entries and the directory entries. This property, was previously leveraged to propose a distributed, duplicated-tag directory in [41].

The same property leads to a theoretical upper limit in the advantages of increasing directory associativity in WC-dir, which results in an ideal directory from the point of view of precision and lack of replacements. In our case this limit is a 1024-way (i.e., with just 2 sets) directory. Note that each of the 128 L2 caches in our configuration has a total of 256 sets (2 048-entry, 8-way L2 caches). This way, the 8 least significant bits of the block address are used for indexing each L2 cache and a maximum of $8 \times 128 = 1024$ different block addresses could be stored in the same set number in all the L2 caches. The 7 least significant bits of the block address are also used for determining the directory bank (home directory) for every block address, and thus, all block addresses mapping the same set in the L2 caches map also the same directory bank (every directory bank would register the sharing information for the block addresses stored in two of these 128×8 global L2 sets). Therefore, a 100%-coverage directory organized as 128 2-set, 1024-way sparse directories removes all

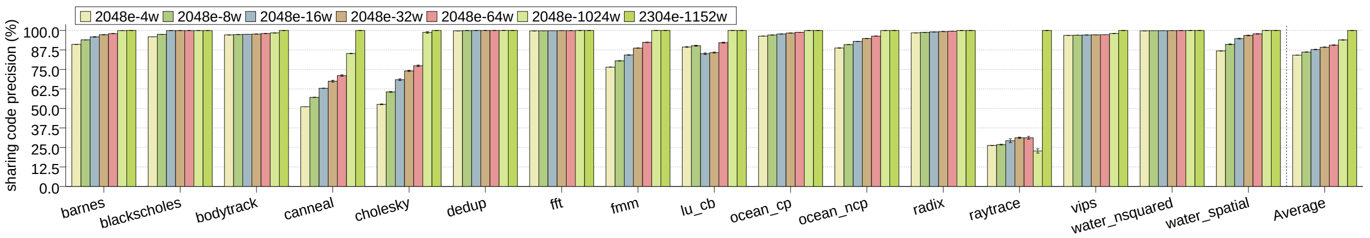


Fig. 13: Precision per address measured as the average for each address of the ratios between the actual number of sharers and the number of sharers encoded in the directory. The directory is sampled every 100 000 cycles. Associativity is varied from 4 ways to 2048 ways.

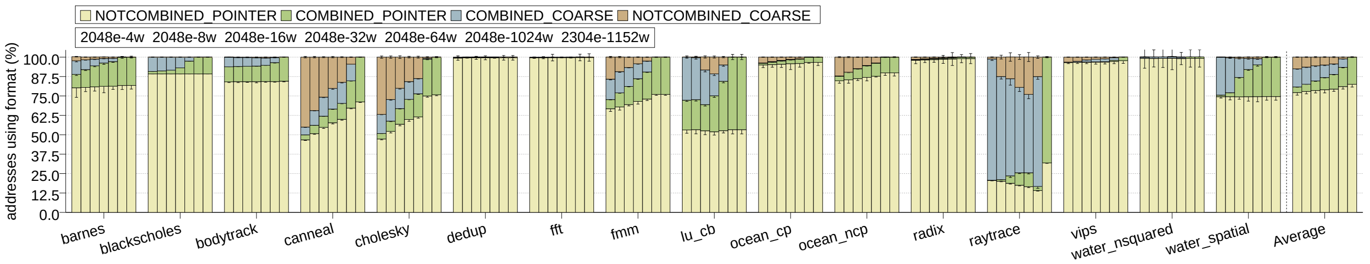


Fig. 14: Entries used in each format per address. The directory is sampled every 100 000 cycles. Associativity is varied from 4 ways to 2048 ways.

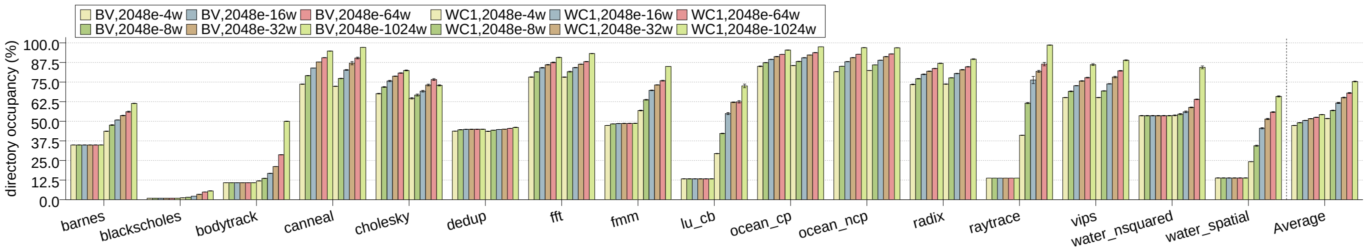


Fig. 15: Directory occupancy with varying directory associativity for WC-dir and BV. The directory is sampled every 100 000 cycles. Associativity is varied from 4 ways to 2048 ways.

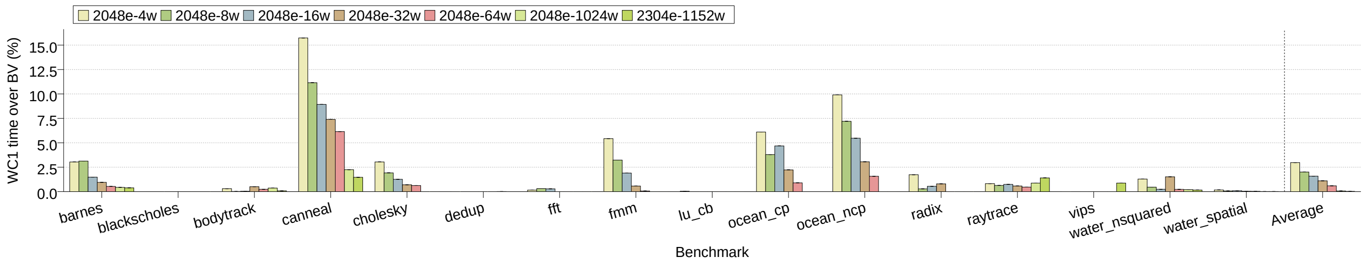


Fig. 16: Increase in the normalized execution time of WC1 with respect to BV with varying directory associativity.

possible directory conflicts and their negative effects (i.e., the necessity to compress the sharing information or to invalidate addresses in the L2 caches due to directory evictions). Such a WC-dir behaves like a duplicated tag directory. Obviously, due to such a large associativity, the directory access time in this case may increase to the point of making it impractical (which is also the main concern with duplicated-tag directories), but we use this configuration point as the theoretical upper limit in associativity for WC-dir.

Fig. 13 shows the precision of WC-dir as the number of ways increases up to the theoretical upper limit (2 sets and 1024 ways per set). For this configuration, perfect precision (100% precision) is reached for most applications, but not for all of them. The reason is that in the simulated cache coherence protocol, L2 replacements are issued to the directory in parallel with the request of the block that causes the replacement. If the request happens to arrive to the directory before the replacement, then the directory may require one extra entry for a short period of time (until the replacement arrives to the directory), provoking the compression of the directory information for some block addresses. This is the reason for the loss of precision observed in some applications and, as shown, it can be significant in some cases[2]. To confirm this

2. Notice that 100% precision could be achieved in these cases if both the replacement message and the request message were issued together, as with the Implicit Replacements technique presented in [42], so that they arrive at the same time to the directory.

end, we added 128 ways to each set of the theoretical upper-limit, such that in the worst case, the L2 cache replacements that arrive before the corresponding requests that cause them never result into compression in the directory. As it can be observed with the `2304e-1152w` bars (i.e., 2 sets of 1024+128 ways), in that case 100% precision is obtained for all applications.

Fig. 14 shows the formats in which the directory information is found during the execution of the applications. The possible formats are: *combined_coarse*, that is a coarse bit vector scheme comprising several ways in the set; *combined_pointer*, that is several ways in the set with all pointers devoted to a single address; *notcombined_coarse*, that is a single way in the set in the coarse bit vector format; and *notcombined_pointer*, that is a single way in the set in the pointer format. Only *combined_pointer* and *notcombined_pointer* store directory information in a precise way. The other two formats generally lead to a loss of precision. As associativity is increased, and therefore conflicts in the directory sets are reduced, there is less need to compress sharing information, and therefore precise formats are employed more often. Again, the race of requests with replacements to arrive to the directory is the reason for the use of the compressed formats in some cases even with 1024 ways.

We analyze in Fig. 15 the evolution of directory occupancy with varying associativity. For comparison purposes, we also present the results for the baseline (the non-scalable bit vector directory). Two observations can be drawn in light of the results shown in Fig. 15: the first is that way combination allows for better use of the limited number of entries available in a typical sparse directory, and the second is that increasing associativity in WC-dir always results in increased occupancy. From these two observations, it can be explained that WC-dir can reach the capabilities of the non-scalable bit vector directory by making the most of the directory entries.

Finally, we show in Fig. 16 the execution times of WC-dir as the associativity of the directory increases to 1152 ways. As this figure aims to show the potential benefit of increasing the number of directory ways, directory access latency has been maintained to 5 cycles for all the configurations. The execution time of each configuration is normalized with respect to a BV directory with the same associativity as WC-dir, which marks the lowest possible execution time in each case. We can see that effectively WC-dir reaches the performance of a non-scalable BV directory as directory associativity is increased. Particularly, from the 64-way configuration on, the performance degradation that is observed is less than 1% on average.

5.4 Varying private cache size and core count

Scaling the 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 [13]. In such scenarios, shared addresses are rare and WC1 would approach very closely the behavior of BV. As the L2 cache size increases, so does sharing (i.e., temporary private addresses turn into shared ones [13]), 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 [13]) 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 increase the number of sharers only for widely shared addresses [13]. 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 performance is significantly lower, and therefore, the advantage of WC1 with respect to LP1 also becomes smaller.

6 Conclusions

This work presents and evaluates *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 [7]. Like other contemporary proposals such as SCD and Pool, 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. In comparison to the Pool directory, WC-dir avoids the need to have to handle an additional structure (the pool of pointers added by Pool). 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 can reach the average execution times of the more expensive SCD and improves over Pool and SCD75 (an implementation of SCD of comparable size). Compared to the non-scalable bit vector sparse directory, we also show that WC-dir can practically meet its performance levels (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 and Pool's for 128 cores, and moreover it remains constant as we increase the number of cores, whereas it grows for SCD, and to a lesser extent, for Pool. The only downside that we have observed for WC-dir is some more extra network traffic. Particularly, WC-dir increases traffic about 5% on average when compared with a similarly sized SCD (SCD75 configuration) and about 10% compared with a SCD configuration with the same number of entries, which requires 43% more area. The difference in network traffic is even lower as compared to Pool (about 3% on average) despite the latter needing also 34% more area.

Observe, however, that the WC1 design evaluated in this work puts the emphasis on minimizing area overhead while maintaining (or closely approaching) the execution time of the non-scalable bit vector directory. 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, that would cut down the traffic penalty whilst still preserving advantages over SCD and Pool (lower execution time, less area —although to a lesser extent— and simpler implementation).

ACKNOWLEDGMENTS

This work has been supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant "RTI2018-098156-B-C53".

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, "Kilocore: A fine-grained 1,000-processor array for task-parallel applications," *IEEE Micro*, vol. 37, no. 2, pp. 63–69, Mar. 2017.

[5] A. Kodi, K. Shifflet, S. Kaya, S. Laha, and A. Louri, "Scalable power-efficient kilo-core photonic-wireless noc architectures," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 1010–1019.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *42nd Int'l Conf. on Parallel Processing (ICPP)*, Oct. 2013, pp. 562–571.

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

[20] R. Titos-Gil, A. Flores, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Way-combining directory: An adaptive and scalable low-cost coherence directory," in *Int'l Conf. on Supercomputing (ICS)*, Jun. 2017, pp. 20:1–20:10.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] R. Fernández-Pascual, A. Ros, and M. E. Acacio, "To be silent or not: On the impact of evictions of clean data in cache-coherent multicores," *Journal of Supercomputing (SUPE)*, vol. 73, no. 10, pp. 4428–4443, Mar. 2017.

[40] 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.

[41] A. Ros, M. E. Acacio, and J. M. García, "A scalable organization for distributed directories," *Journal of Systems Architecture (JSA)*, vol. 56, no. 2-3, pp. 77–87, Feb. 2010.

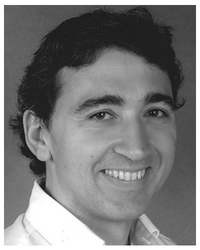
[42] ——, "Scalable directory organization for tiled CMP architectures," in *Int'l Conference on Computer Design (CDES)*, Jul. 2008, pp. 112–118.

Rubén Titos-Gil received MS and PhD degrees in Computer Science from the University of Murcia, Spain, in 2006 and 2011, respectively. As a PhD student, he was awarded a FPU scholarship from the Spanish Government. After holding post-doctoral positions at Chalmers University of Technology, Sweden, and at the Barcelona Supercomputing Center, Spain, in 2015 he rejoined the University of Murcia where he has since served as an adjunct professor. His research focuses on hardware support for synchronization in parallel processors, with an emphasis on transactional memory.

Antonio Flores Antonio Flores received the PhD degree in Computer Science from the Universidad de Murcia, Spain, in 2010. Currently, he is an Associate Professor in the Computer Engineering Department at the Universidad de Murcia. From June 2018, he serves as Dean of the Faculty of Computer Science at the Universidad de Murcia. His research interests include CMP architectures, processor microarchitecture, and power-aware cache coherence protocol design.

Ricardo Fernández-Pascual received the MS and PhD degrees in computer science from the Universidad de Murcia, Spain, in 2004 and 2009, respectively. In 2004, he joined the Computer Engineering Department as a PhD student with a fellowship from the regional government. In 2006, he joined the Computer Engineering Department of the Universidad de Murcia where he is currently an associate professor. His research interests include general computer architecture, memory hierarchies for parallel processors, and performance simulation.

Alberto Ros Alberto Ros is Associate Professor at the University of Murcia, Spain. He received the Ph.D. degree in computer science from the same university, in 2009, after being granted with a fellowship from the Spanish government to conduct the Ph.D. studies. He hold post-doctoral positions at the Universitat Politècnica de València and at Uppsala University. He has co-authored more than 60 research papers in international journals and conferences. His research interests include cache coherence protocols, memory hierarchy designs, and memory consistency for multicore architectures.



Salvador Petit Salvador Petit (M'07) received the PhD degree in computer engineering from the Universitat Politècnica de València (UPV), Spain. Since 2009, he has been an associate professor with the Computer Engineering Department, UPV, where he has been teaching several courses on computer organization. He has authored more than 100 refereed conference and journal papers. His current research interests include multithreaded and multicore processors, memory hierarchy design, task scheduling, and real-time systems. He is a member of the IEEE and the IEEE Computer Society. In 2013, he received the Intel Early Career Faculty Honor Program Award.

Julio Sahuquillo Julio Sahuquillo (M'04) received the BS, MS, and PhD degrees from the Universitat Politècnica de València, Spain, all in computer engineering. He is a full professor with the Department of Computer Engineering, Universitat Politècnica de València. He has taught several courses on computer organization and architecture. He has authored more than 120 refereed conference and journal papers. His current research interests include multi- and manycore processors, memory hierarchy design, cache coherence, GPU architecture, and architecture-aware scheduling. He is a member of the IEEE and the IEEE Computer Society.

Manuel E. Acacio is a Full Professor of computer architecture and technology at the University of Murcia, Spain. Dr. Acacio obtained his PhD degree in Computer Science in March 2003. Before, in the summer of 2002, he worked as a summer intern at IBM TJ Watson, Yorktown Heights (NY). Currently, Dr. Acacio leads the Computer Architecture & Parallel Systems (CAPS) research group at the University of Murcia. He is author of about 100 papers in refereed international conferences and journals. As well, he has served as a committee member of important conferences, ICPP and IPDPS among others. His research interests are focused on the architecture of multiprocessor systems. From April 2011 to April 2015, Dr. Acacio served as an associate editor of IEEE Transactions on Parallel and Distributed Systems Int'l Journal, since August 2016 he is member of the editorial board of MPDI Computers Int'l Journal, and more recently, since September 2018, he serves as academic editor in the editorial board of Hindawi Scientific Programming journal. He is also member of the board of distinguished reviewers of ACM Transactions on Architecture and Code Optimization Int'l Journal since May 2014.