Noname manuscript No.
(will be inserted by the editor)

Are Distributed Sharing Codes a Solution to the Scalability Problem of Coherence Directories in Manycores? An Evaluation Study

Ricardo Fernández-Pascual · **Alberto Ros** · **Manuel E. Acacio**

Received: date / Accepted: date

Abstract The development of efficient and scalable cache coherence protocols is a key aspect in the design of manycore chip multiprocessors. In this work, we present a comprehensive evaluation of a kind of cache coherence protocols that, despite having been already implemented during the 1990s for building large-scale commodity multiprocessors, have not been considered in the context of chip multiprocessors yet.

In particular, we evaluate two directory-based cache coherence protocols based on the idea of having the sharing code of each memory block distributed between the different sharers (distributed sharing code). The first one employs simply-linked lists to encode the information about the sharers of the memory blocks, whilst the second one does the same using doubly-linked lists, which improves the management of replacements. We compare these two organizations with three protocols that use centralized sharing codes, each one having different directory memory overhead: one of them implementing a non-scalable bit-vector sharing code and the other two implementing more scalable limited-pointer schemes with one and two pointers respectively.

Simulation results show that for large-scale chip multiprocessors, the protocol based on distributed doubly-linked lists dramatically reduces the memory overhead of a non-scalable bit-vector directory, while at the same time it achieves its performance levels. This is achieved with just a small degradation on dynamic energy consumption (approximately 10% on average). This way, our results point out that for manycores, coherence directories based on distributed sharing codes are appealing alternatives to contemporary coherence directories based on centralized sharing codes.

Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain
E-mail: {rfernandez,aros,meacacio}@um.es

1 Introduction

As the number of transistors in a chip increases following the well-known Moore's law, so it does the number of processor cores implemented in chip multiprocessors (CMPs). Very far are those designs integrating a small number of cores, such as the pioneer dual-core IBM POWER4 [1]. Nowadays technology enables the orchestration of manycores, that is CMPs architectures with dozens of integrated cores, and several examples are already being commercialised, such as the 72-core Intel's Knights Landing [2] and Tilera's Tile GX8072 [3]. This trend towards more and more cores, however, has not yet reached its end and manycores with hundreds of cores are expected to become a reality in the near future [4].

Design decisions about communication and synchronization mechanisms among cores in these *densely populated* manycore architectures become a key aspect for the final performance of the multicore. If the current trend continues, future manycores will go on providing the shared memory model as the low-level interface and will rely on a cache coherence protocol implemented in hardware to ensure coherence between data stored in every core's private caches [5]. This way, both communication and synchronization (the latter usually implemented through normal load and store instructions to shared addresses) require a properly designed cache coherence protocol to achieve expected high performance levels.

In systems with a large number of processing cores, directory-based cache coherence protocols appear as the only viable alternative. In these protocols, a directory structure which is physically distributed among the different cores (or group of cores, depending on the particular implementation) is responsible for keeping track of the identity of the sharers of every memory block residing in one or several of the private caches. This way, every memory block is assigned to a single directory bank and all cache misses from the private caches for that block are sent to it. On receiving a cache miss, the directory information for the particular block is retrieved, and based on that, coherence actions (if needed) are carried out so that the cache miss can be resolved.

The way the directory structure codifies the set of sharers for every memory block determines the amount of extra memory required for this structure (directory memory overhead) and ends up limiting the range of cores at which cache coherence can be provided in a practical way. Codifying the set of sharers using structures whose size per node depends linearly on the number of cores is certainly guarantee of non scalability. For example, the well-known bit-vector sharing code (that requires one bit per core) could be employed in an architecture with a handful of cores but it is absolutely incompatible with scalability. Besides directory memory overhead, we can also identify two other aspects of a coherence directory that could restrict its scalability. Both aspects are closely related with the size of the coherence directory. First of all is the amount of coherence traffic generated on every coherence event (number of invalidations or cache-to-cache transfer orders). One approach for reducing directory memory overhead is by means of using in-excess representations of the

set of sharers (more sharers than necessary are typically included). Of course, less precision in the sharing code means more *false* sharers, and thus, more coherence messages per coherence event, which also limits scalability [6]. An extreme example is the AMD's Hammer cache coherence protocol, which did not dedicate any bits to the sharing code, which resulted into broadcasting coherence messages on every coherence event [7]. Finally, the other important aspect that usually goes unnoticed is the amount of time required to extract the identify of the sharers on every coherence event. Some approaches save directory memory by codifying the set of sharers in a way that several directory cycles are required. Even in some cases the total number of cycles is not always the same [8]. This affects the occupancy of the directory controller, and consequently, the latency of the cache miss causing the access to the directory, and also the latency of other misses waiting for processing at the directory. Obviously, these three aspects (directory size, generated network traffic and amount of work needed to *unpack* directory information) determine the energy efficiency of the cache coherence protocol.

The design of scalable coherence directories for systems with a large number of cores has been extensively studied for traditional multiprocessors. In that context, the most scalable protocols —those which kept sharing information in a directory distributed among nodes— were classified in two categories [9]: *memory-based* schemes and *cache-based* schemes. Memory-based schemes store the sharing information about all the cached copies of each block in a single place, which is the home node of that block. In traditional multiprocessors, the home node was associated with the main memory, and that is why they were called memory-based schemes. On the other hand, in cache-based schemes not all the sharing information about a single block is stored in the home node. Instead, it is distributed among the caches holding copies of the block while the home node only contains a pointer to one of the sharers. Usually, one or two pointers are stored along with each copy of the block, forming a distributed linked list of sharers.

Nowadays, current cache coherence proposals for manycore architectures assume centralized directory schemes. In the context of multicore architectures, the term *memory-based* for referring to these approaches is no longer appropriate, since the home node is now typically associated with the last level cache (LLC) in the chip, not with main memory. Hence, we will use instead the term *centralized sharing code*. On the other hand, although distributed schemes were employed in several commodity multiprocessors during the 90s ([10–13]), they have not been analyzed in the context of multicore architectures. The main advantage of these schemes, which we will call *distributed sharing code* schemes, is that they allow for implementations with lower directory memory overhead than the centralized sharing code ones with the same precision [9]. However, they show several disadvantages, such as higher cache miss latency since several messages are needed to *discover* the identity of all the sharers, some modifications that must be introduced in the private caches, and the increased complexity for managing cache evictions.

In this manuscript, we extend our previous work [14] and present a comprehensive evaluation of two coherence directory implementations employing distributed sharing codes (*SingleList* [15] and *DoubleList* [16]) putting special emphasis on scalability. *SingleList* is based on the use of simply-linked lists, thus being the lowest-overhead coherence directory that can be designed with a distributed sharing code. On the other hand, *DoubleList* employs doubly-linked lists and resembles the coherence protocols based on distributed sharing codes implemented in several commodity multiprocessors [11–13]). The evaluation is done considering resulting performance, memory overhead, network traffic and energy consumption. To the best of our knowledge, this is the first work evaluating these two distributed sharing code schemes in the context of manycores.

For comparison purposes, we compare these two designs against three implementations of a coherence directory using centralized sharing codes (*BitVector* [17], *OnePointer* and *TwoPointers* [18]). *BitVector* employs non-scalable bit-vectors (full-map), and thus, entails significant directory memory overhead. This implementation, however, incurs both lowest network traffic levels and reduced latency cache misses. *OnePointer* and *TwoPointers* ensure scalable directory memory overhead by means of using a limited pointers scheme with 1 or 2 pointers, respectively. Obviously when the number of pointers is not enough to codify all the sharers of a memory block, the directory resorts to broadcasting, which hurts network traffic and increases cache miss latencies. All the protocols use the MESI states and behave as similarly as possible in all other aspects. Compared with our previous work [14], here we add results for two interesting design points (*TwoPointers* and *DoubleList*), and significantly extend the evaluation considering the impact of the coherence directory on energy consumption and using a more detailed network model.

Through detailed simulations of 16-core and 64-core CMPs, we show that all the configurations obtain similar results in terms of performance and energy consumption when the number of cores is not so big (16 cores). However, important differences between the alternatives appear when the number of cores is increased to 64, underscoring the significant incidence that the coherence directory may have in future manycores. In particular, we observe that for large core counts *SingleList* obtains worse performance and energy consumption figures (approximately 13% and 20% on average, respectively) than *BitVector*. We find that the reason for this performance degradation is the increased contention that the *SingleList* protocol introduces at the level of the directory controller. This is due to excessive locking time for updating the list of sharers upon cache misses and evictions. We also find out that *DoubleList* can solve this problem thanks to the additional pointers included at the private cache level and meets the performance results of *BitVector* with just a small increase in dynamic energy consumption (about 10% on average). This is achieved with an average reduction in leakage energy of 13%. Of course, this comes at the expense of increased requirements in terms of directory memory overhead with respect to *SingleList*. This way our results demonstrate that for future manycores, coherence directories based on distributed sharing

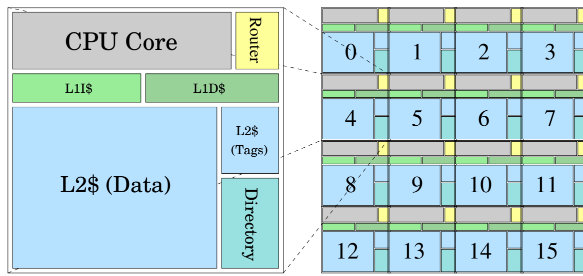


Fig. 1: Architecture of a tiled CMP

codes are appealing alternatives to contemporary coherence directories based on centralized sharing codes.

The rest of the manuscript is organized as follows. We start by presenting in Section 2 the baseline architecture we will assume along this work. Then, in Section 3 we describe the behavior of two implementations of a cache coherence protocol using two distributed sharing codes. Subsequently, in Section 4 we compare the directory memory overhead for the five configurations evaluated in this work (*BitVector*, *OnePointer*, *TwoPointers*, *SingleList* and *DoubleList*). Section 5 presents our simulation environment and detailed results in terms of execution time, network traffic, and energy consumption for the different configurations are shown and analyzed in Section 6. Finally, Section 7 contains the main conclusions of this work.

2 Baseline Architecture

In this work we assume single CMP systems built using a number of tiles [19]. Tiled chip-multiprocessors are designed as arrays of identical or close-to-identical building blocks (tiles). In these architectures, each tile contains a processor core, private L1 data and instruction caches, and a bank of the L2 cache. The L2 cache is logically shared by all cores, but it is physically distributed among tiles. As in current Intel-style multicore architectures [20], we assume inclusive caches, that is L1 caches' content is included in L2.

Memory blocks are distributed among the different banks of the shared L2 cache by using a physical mapping policy. Particularly, we use the less significant bits of the block address to define the home bank for every block [21, 22]. This way, blocks are assigned to L2 cache banks in a round-robin fashion with block-size granularity. Since the L2 cache is the last level cache in our baseline architecture we will refer to this cache either as L2 or as LLC.

Each tile also has its network interface to connect to the 2D mesh on-chip interconnection network. Figure 1 shows the organization of the tiled CMPs we model in this work when the total number of cores is 16.

Private L1 caches are kept coherent by means of a directory-based cache coherence protocol that implements MESI states. The directory structure is

distributed between the L2 shared cache banks, usually within the tags' portion [23]. This way, each tile keeps the sharing information of the blocks mapped to the L2 cache bank that it contains. This sharing information comprises two main components (apart from other implementation-dependent bits): the *state bits* used to codify one of the three possible states the directory can assign to the block (*Uncached*, *Shared* and *Private*), and the *sharing code*, that holds the list of current sharers (*centralized sharing code*) or a pointer to one of the sharers (*distributed sharing code*). Most of the bits of each directory entry are devoted to codifying the sharing code, and in this work we consider three centralized sharing codes (*BitVector*, *OnePointer*, and *TwoPointers*) and two distributed sharing codes (*SingleList* and *DoubleList*). This aspect is the one which distinguishes the different configurations evaluated in Section 6.

3 Coherence Protocols Based on Distributed Sharing Codes

The main difference between the linked protocols considered and evaluated in this work (*SingleList* and *DoubleList*) and those based on a coherence directory using centralized sharing codes is that the list-based ones store directory information for each memory block in a distributed way. Particularly, the home node in list-based protocols stores the identity of only one of the possibly many sharers of the memory block. The rest of sharers is represented using a linked list constructed through pointers in each of the L1 cache entries.

In the case of *SingleList*, each L1 cache entry needs only one pointer which stores the identity of the next sharer in the list or the null pointer if the current node is the last element in the list. In the case of *DoubleList*, each L1 cache has two pointers, one storing the next node (or null for the last node) and another storing the previous one (or null for the first node). In all cases, the null pointer is represented by codifying the identity of the sharer itself (i.e., the node points to itself).

Therefore, directory information for each memory block in these protocols is distributed between the home node and the set of sharers. As it will be shown in Section 4, the fact that most of the directory storage is moved to the L1 caches (which are much smaller than the L2 cache) brings important advantages like reduced requirements of the directory structure in terms of memory overhead (and thus, energy consumption) and improved scalability.

However, since directory information is stored in a distributed way in the list-based protocols, several messages are required between the sharers and the home node to update this information. Some of these messages would not be needed in a directory protocol using a centralized sharing code. Besides, list traversal operations through control messages traveling on the interconnection network are needed to discover the identity of all sharers. This can lead to increased write miss latencies due to invalidation of the sharers takes place sequentially.

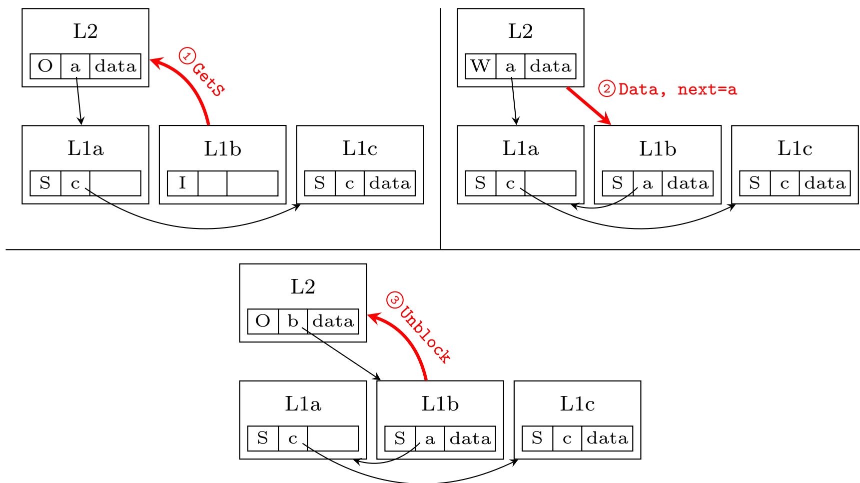


Fig. 2: Example of a read miss in *SingleList*

3.1 *SingleList*: A simply-linked list protocol

Updates to the list of sharers in the *SingleList* protocol are always serialized by the home node, which remains blocked (i.e., other requests for this memory block are not attended) until the modification of the list structure has been completed. This way, we guarantee that two or more update operations cannot take place simultaneously. There are three transactions that modify the linked list: read misses, write misses, and replacements. Next sections explain these modifications.

3.1.1 How read misses are managed in SingleList

The procedure to resolve read misses for uncached data (i.e., when the memory block is not held by any of the private caches) is almost identical in the *SingleList* protocol and the *BitVector* protocol: once the request (read miss) reaches the corresponding home L2 bank, it sends back a message with the memory block to the requester, which subsequently responds with the *Unblock* message to the directory. The home L2 bank uses the pointer available in the tags' part of the L2 cache to store the identity of the only sharer up to the moment.

When no L1 cache nor the home L2 bank keep a copy of the requested memory block, the directory controller will send a request to memory and once data is received, it will be stored in the L2 cache and a copy of the memory block will be sent to the requester. In this case, the memory block will be put in the `E` (Exclusive) state, or `M` (Modified) in case of a write miss, in the private cache that suffered the miss.

The main difference between the *SingleList* and *BitVector* protocols with respect to read misses is observed when the block is found as *shared* at the home L2 cache bank. This scenario illustrated in Figure 2 assuming two sharers. Upon receiving the read request `(1)`, the L2 sends to the requester both the memory block and the identity of the first sharer in the list `(2)`. Then, the requester stores the memory block in its L1 and sets up the pointer field in the corresponding entry to the identifier included in the L2 response message (its *next sharer*). After this, it sends an *Unblock* message to the L2, which overwrites the pointer field with the identity of the requester `(3)`. This way, the list structure keeps the identity of the sharers of a particular memory block in reverse order to how read misses were processed by the L2.

If, on the contrary, the memory block is found in `private` state in the L2 (i.e., it is either in `E` or in `M` state in only one of the L1 caches), the read miss is forwarded by the directory controller to the only L1 cache that holds a valid copy of it. Upon receiving the forwarded request, the corresponding L1 cache responds directly to the requester with a message containing the memory block and its own identity, which will be used by the requester to update its *next sharer* field. Then, the requester proceeds just like in the previous case.

As it can be observed, updates of the list structure used to keep the identity of all the sharers of every memory block do not need to introduce any new messages in the *SingleList* protocol with respect to *BitVector*. This is because response messages are used to transport all the information (one identifier in this case) required to maintain the list structure.

3.1.2 How write misses are managed in SingleList

Invalidation-based protocols, such as the ones considered in this work, resolve write misses by invalidating all the copies of the memory block held by the L1 caches. The corresponding directory controller at the L2 starts the invalidation process in parallel with sending a response message with the data block back to the requester.

On a write miss, in a traditional directory protocol with a centralized sharing code (such as *BitVector*), the directory controller at the corresponding home L2 cache bank sends one invalidation message to each one of the sharers. This is possible because all the available information about the sharers is stored at the home L2 cache bank (although it may be inexact as in the *OnePointer* and *TwoPointers* protocols). Therefore, invalidation messages are sent in parallel (although if the interconnection network does not provide multicast support they would be created and dispatched by the directory controller sequentially).

On the contrary, the invalidation procedure in a directory protocol with a distributed sharing code (such as *SingleList*) must be done serially. In this case, the L2 only knows the identity of one of the sharers, which in turn knows the identity of the next one, and so on. This way, invalidation messages must be created and sent one after another, as the list structure is traversed. Once the last sharer is reached, a single acknowledgement message is sent

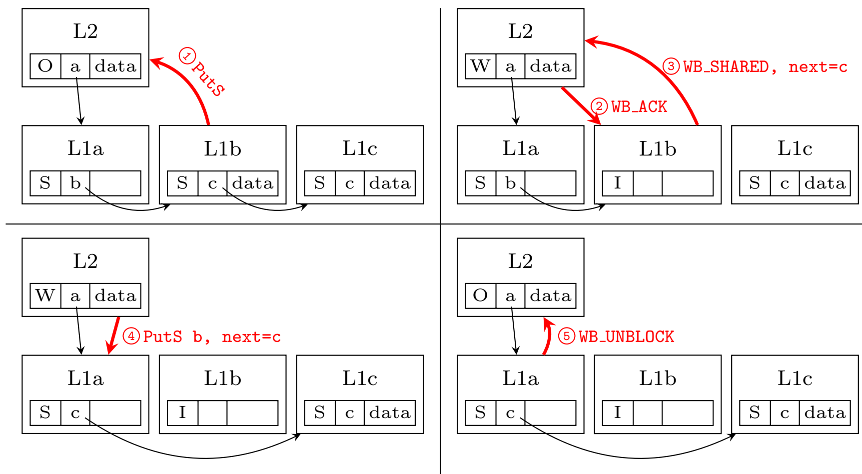


Fig. 3: Example of shared block replacement in *SingleList*

to the requester as a notification that all the copies in the L1 caches have been deleted. As it can be noted, the latency of write misses is therefore increased, especially for widely shared memory blocks. But this also brings one advantage: whereas in the *BitVector* protocol all invalidation messages entail the corresponding acknowledgement response, in the *SingleList* protocol just one acknowledgement is required. This obviously reduces network traffic when the number of sharers is large.

The data block on a write miss is sent just like in the case of a read miss and exactly like *BitVector*, taking into account whether the block is in *private* state at the L2 and hence needs to be sent by the L1 which currently has it after receiving the invalidation from L2 and along with its acknowledgement; or in *shared* state, in which case the data is sent by the L2.

For both the *BitVector* and *SingleList* protocols, the requester sends the *Unblock* message to the home L2 cache bank only when the invalidation process has finished (it has collected all the acknowledgements to the invalidation messages sent by the directory controller in the case of the *BitVector* protocol, or the only acknowledgement response that is needed in the *SingleList* one) as well as the response with data has arrived. As in the case of read misses, upon receiving the *Unblock* message the directory controller takes note of the new holder of the memory block using the pointer available at the L2 cache.

3.1.3 How replacements are managed in SingleList

Replacements of data blocks from L1 caches in `E` or `M` state (*private* in the L2) proceed exactly the same way in both *SingleList* and *BitVector* protocols. In these cases, the L1 sends a request to the L2 asking for permission, and upon receiving authorization from the L2, the L1 sends an acknowledgement (if `E`) or the modified data block (if `M`) to the L2. By requiring the L1 to ask for

authorization before sending the replaced data to the L2, both protocols avoid some race conditions that complicate their design (and that, if not correctly addressed, would lead to deadlocks).

However, the main difference between the *SingleList* and *BitVector* protocols has to do with the management of replacements of `S` (Shared) data. This is because in a protocol with a centralized sharing code replacements of shared data are typically performed in a silent way (the replaced line is simply discarded and no message has to be sent to the L2). Although not sending replacement hints for shared data could lead to the later appearance of some unnecessary invalidation messages, previous works have demonstrated that shared replacements are preferable to the waste of bandwidth and increase in the occupancy of cache and directory controllers that otherwise would be suffered [24, 25]. This is especially true when the number of cores is large.

Differently, the *SingleList* protocol cannot implement silent replacements since the list structure has to be correctly maintained after a replacement. Thus, the replacement process requires involving the L2 and some L1s. This process is depicted in Figure 3, which assumes that there are two other sharers apart from the replaced block. Before a shared memory block can be replaced, a replacement request is sent to the L2 `(1)`. When the L2 receives the request and it is ready to handle it, it sends a message authorizing the replacement `(2)`. This message is answered with another that carries the value of the pointer field kept at the L1 cache which stores the identity of the following L1 cache in the list of sharers `(3)`. After sending that message, the L1 cache can discard the memory block and all its metadata from its cache. It will not be contacted again regarding this transaction. Upon receiving the message with the following sharer of the replacing node, the L2 needs to update the sharing list (which is momentarily disconnected because the replacing node has discarded its pointer). For this, if the identity of the replacing node coincides with the sharer stored at the L2 (i.e., the replacing node is the first sharer of the list), then the value of the pointer at the L2 is changed to the identity of the following node included in the last message received from the requester. Otherwise, the L2 cache forwards the replacement request to the sharer codified in its pointer field `(4)`. The message keeps propagating through the list of sharers until the node that precedes the replacing node in the list is reached. This is identified because its pointer to the next sharer will match the requester identity. At this point, the pointer in the preceding node is updated with the information included in the message (the identity of the node following the replacing node), reconnecting the list of sharers. After updating its next pointer, the node sends an acknowledgement to the L2 and the operation completes `(5)`.

Notice that the identity of the next sharer of the replacing node cannot be sent along with the first message of the transaction, because if the L2 receives another request for that line from a different node before the replacement request arrives, the information may become obsolete. The L2 will not attend other requests for that line while the replacement is being performed to avoid concurrent modifications to the list of sharers.

As we will show next, the fact that replacements for shared data in the *SingleList* protocol cannot be done silently significantly increases the number of messages on the interconnection network (bandwidth requirements) and, what is more important, the occupancy of the directory controllers at the L2 cache. It is important to note that although write buffers are used at the L1 caches to prevent delaying unnecessarily the cache miss that caused the replacement, the fact that the directory controller "blocks" the memory block being replaced results in longer latencies for subsequent misses to the replaced address.

3.2 *DoubleList*: A doubly-linked list protocol

The advantage that the *DoubleList* protocol has over the *SingleList* protocol is that replacements of shared data can be done without traversing the list of sharers and without involving the L2 in many cases. These replacements are not completely silent as in the case of the *BitVector* protocol, but they usually involve fewer messages than replacements in the *SingleList* and, more importantly, the L2 does not need to be blocked. In fact, with this protocol a node can be removed from the list of sharers while another node is being added.

3.2.1 How read misses are managed in DoubleList

For uncached data, the procedure is the same in *DoubleList* as in *SingleList* and *BitVector*. *DoubleList* behaves also almost the same as *SingleList* when the memory block is found in the *private* state at the L2. The only difference is that, when the L1 holding the block receives the request forwarded from L2, it will update its previous sharer pointer to point to the new sharer before responding to the requester. The requester's next pointer will be updated to point to the former owner (as in *SingleList*) while both the previous pointer of the new sharer and the next pointer of the former owner will be kept as null.

On the other hand, when one sharer already exists, the procedure requires two more messages than in *SingleList*. When the L2 receives the read request, it answers to the requester with the data and the identity of the current first sharer of the list. The requester, which becomes the new first sharer, will use this information to update its next sharer pointer, as in *SingleList*. But additionally, the L2 will send another message to the former first sharer to update its previous sharer pointer with the identity of the requester. After updating its pointer, the former first sharer will send an acknowledgement to the requester. When the requester has received both the data from L2 and the acknowledgement from the former first sharer it will send an *Unblock* message to the L2 to finalize the transaction.

Notice that the extra messages to update the previous sharer pointer of the former first sharer are out of the critical path of the miss (the requester

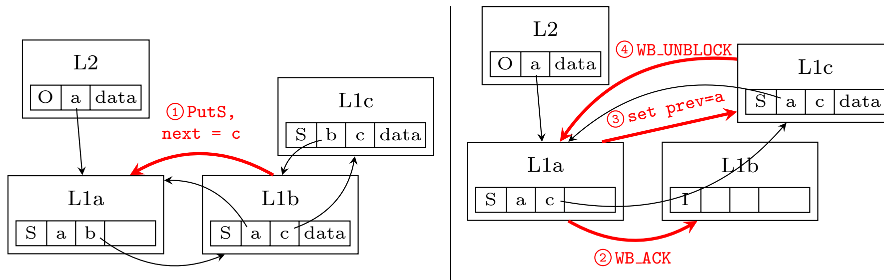


Fig. 4: Example of shared block replacement in *DoubleList*

can use the data as soon as it receives it from L2), although they will slightly increase traffic and may keep the memory block locked by the L2 a bit longer.

3.2.2 How write misses are managed in DoubleList

For handling write misses, the pointer to the previous sharer is not used and the *DoubleList* protocol behaves always exactly the same as *SingleList*.

3.2.3 How replacements are managed in DoubleList

Replacement of private blocks in *DoubleList* works exactly as in *SingleList* and the other protocols. However, thanks to the pointer to the previous node of each sharer, shared replacements in *DoubleList* can be implemented without involving the L2 for all sharers except the first one of the list. Moreover, while *SingleList* has to traverse the list of sharers from the beginning in order to remove the replacing node, *DoubleList* can remove a node contacting only the previous sharer.

The process works as depicted in Figure 4. A node that needs to replace a block and that is not the first sharer in the list will send a request to the node pointed by its previous pointer which includes the identity of its next sharer (1). Upon reception, the previous node needs to check that the requester is still its next sharer because other requests may have changed it while the replacement request was in the network. If not, it will answer with a nack message and the requester will try again. Otherwise, the previous node updates its next pointer to point to the node that was the next sharer of the replacing node. Then, it sends a writeback acknowledgement to the replacing node (2), which discards the block and its metadata upon receiving it, and another message to its new next sharer (unless there isn't any) to request it to update its previous sharer pointer (3). The previous sharer will stay partially blocked until it receives a response from its new next sharer (4) to avoid deadlocks and incorrect list updates. While partially blocked this way, the cache will only attend to loads from its processor and requests to update its previous sharer pointer, but not to other replacement requests, invalidations from the L2 or writes from its processor.

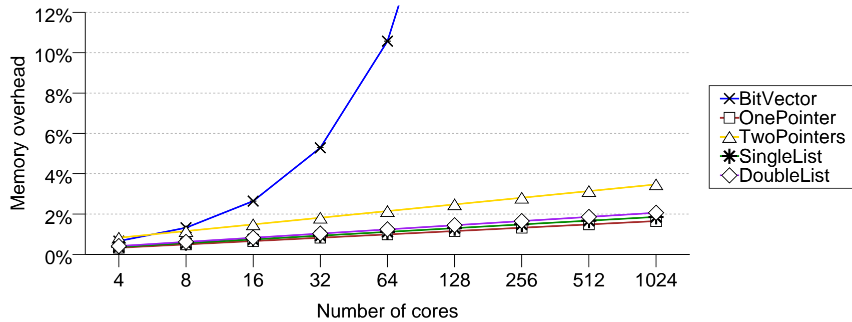


Fig. 5: Memory overhead of the evaluated protocols.

In the case that the node that needs to replace is the first sharer of the list, it will send its replacement request to the L2. The L2 will update its pointer to the first node, send the writeback acknowledgement to the requester and, if there was a second sharer, send a message to it to set its previous pointer to null. The L2 will be blocked until it receives a response from the new first sharer (if any).

Notice that replacements of shared L1 blocks in *DoubleList* do not usually involve the L2 and can occur concurrently to list insertions and other replacements.

4 Directory Memory Overhead Analysis

One of the reasons why directory protocols based on a distributed sharing code were popular two decades ago was their good scalability in terms of the amount of memory required to store sharing information. In the end, this results into lower area requirements and, what is more important nowadays, better scalability in terms of static power consumption. Whereas the amount of memory required per directory entry with a bit-vector sharing code (as the one used in the *BitVector* protocol) grows linearly with the number of processing cores (one bit per core), it grows logarithmically for the rest of protocols analyzed in this paper. *SingleList* and *DoubleList* need some additional information in each L1 entry (pointers), but this is not a problem for scalability because the number of entries in the L1 caches is always much smaller than in the L2 cache banks.

Figure 5 compares the directory protocols considered in this work in terms of the memory overhead each one of them introduces, assuming 4-way L1 caches with 128 sets and 16-way L2 caches with 256 sets per core and 64 byte blocks. Particularly, we measure the percentage of memory (in bits) added by each protocol with respect to the total number of bits dedicated to the L1 and L2 caches. The figure shows that, as expected, the scalability of the *BitVector* protocol makes it practical only in configurations with a small number of cores.

To improve memory scalability, *OnePointer* replaces the bit-vector used in each of the L2 cache entries of *BitVector* with a limited pointer sharing code with only one pointer for the first sharer. If more than one node is sharing the block, a different state is used and invalidations will be broadcast to all nodes. In this case, the number of bits per directory entry grows as $\log_2 N$, being N the total number of cores. As it will be analyzed in the next sections, this scalability in terms of area comes with a performance degradation due to increased interconnection network traffic.

In order to mitigate the traffic increase due to broadcasts observed in *OnePointer*, *TwoPointers* uses two pointers for the two first sharers of the block, and an *overflow* bit to codify when more than two caches are sharing the block and invalidations need to be broadcast. *TwoPointers*'s per directory entry memory requirements grow as $1 + 2 \times \log_2 N$.

SingleList sharing information is stored as a pointer in each L2 entry and another one in each L1 entry. Since L1 caches have much fewer entries that L2 caches, the scalability of the *SingleList* in practice is very close to that of *OnePointer*. Finally, *DoubleList* uses two pointers in each L1 instead of one and, therefore, uses more memory than *SingleList*, although is scales similarly.

5 Evaluation Environment

We have evaluated two list-based directory cache coherence protocols comparing them to the other state-of-the-art cache coherence protocols mentioned in this work by using the PIN [26] dynamic binary instrumentation tool and GEMS 2.1 [27] simulator, which have been connected in a similar way as proposed by Monchiero *et al.* [28]. PIN obtains every data access performed by the applications as well as the synchronization skeleton, while GEMS models the memory hierarchy and calculates the memory access latency for each processor request. The interconnection network is modeled with the SiCoSys [29] simulator, which we have connected with GEMS. The simulated architecture corresponds to a single chip multiprocessor (*tiled*-CMP) with either 16 or 64 cores. The most relevant simulation parameters are listed in Table 1. We use the CACTI-P tool included in McPat 1.2 [30] to estimate access time, area requirements and power consumption of the different cache structures assuming a 22 nm technology node and a 3.5 GHz processor frequency, accounting for the precise metadata requirements of each protocol. To estimate the dynamic energy consumption of the interconnection network, we assume that it is proportional to the data transferred [31] and that each flit transmitted through the network consumes the same amount of energy as reading one word from an L1 cache each time that it crosses a link. A flit is the amount of data transmitted in one cycle, and messages between more distant nodes will require more flit retransmissions and hence they will consume more energy.

In order to evaluate and compare the coherence protocols studied in this work, we have implemented in GEMS a traditional directory-based cache coherence protocol (called *BitVector*) using one bit-vector per each memory

Table 1: System parameters.

Memory parameters (GEMS)	
Block size	64 bytes
L1 cache (data & instr.)	32 KiB, 4 ways
L1 access latency	1 cycle
L2 cache (shared)	256 KiB/tile, 16 ways
L2 access latency	6 cycles plus network
Cache organization	Inclusive
Directory information	Included in L2
Memory access time	160 cycles
Network parameters (SiCoSys)	
Topology	2-D mesh (4×4 or 8×8)
Switching technique	Wormhole
Routing method	X-Y determinist
Multicast/Broadcast	Not supported
Message size	4 flits (data), 1 flit (control)
Routing time	1 cycle
Switch time	1 cycle
Link time	2 cycles
Buffer size	6 flits
Link bandwidth	1 flit/cycle

block, a protocol (called *OnePointer*) that uses a single pointer to the owner as sharing information similarly to AMD's *MagnyCours* [32] protocol, another protocol (called *TwoPointers*) similar to the previous one but with two pointers for the two first sharers as sharing information, the two protocols described in Section 3 which use a distributed sharing code implemented by means of linked lists with either a single pointer or two pointers at the private caches (which we have called *SingleList* and *DoubleList*, respectively).

We have conducted simulations using a large number of representative applications from both the SPLASH-2 [33] and the PARSEC 2.1 [34] benchmark suites. *Barnes*, *Cholesky*, *FFT*, *Ocean*, *Radix*, *Raytrace*, *Volrend*, and *Water-NSQ* are from the SPLASH-2 suite and have been configured with the input sizes used in the SPLASH-2 paper. *Bodytrack*, *Canneal*, *Streamcluster*, and *Swaptions* are from the PARSEC 2.1 suite and use the *simmedium* input sizes. We have accounted for the variability of parallel applications as discussed in [35]. To do so, we have performed a number of simulations (at least four) for each application and configuration inserting random variations in each main memory access. All results in this work correspond to the parallel part of the applications.

6 Evaluation Results

This section compares the two list-based cache coherence protocols described sin Section 3 against three implementations of the coherence directory that use several well-known centralized sharing codes. We show simulation results for both execution time and energy consumption of the considered applications. In order to better understand these results, we first analyze for each protocol and sharing code the number of invalidations that are performed per write miss, the

L1 cache miss latency considering where it is spent, the type of replacements in the L1 caches, and the network traffic generated by each coherence protocol. All the experiments have been carried out for configurations with both 16 and 64 cores.

6.1 Number of potential sharers upon write misses

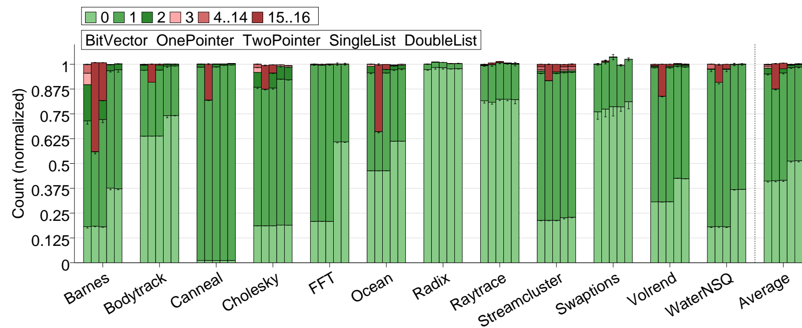
One of the main characteristics of applications impacting the performance of a cache coherence protocol is the number of cores that will receive invalidation messages upon a write miss. The design of both the sharing code and the cache coherence protocol affects this number. For example, a coherence directory that uses a sharing code with a single pointer will have to send invalidation messages to all private caches every time a block shared by two or more cores is written (or replaced from L2). In addition, coherence directories using centralized sharing codes usually employ silent evictions of shared blocks. Silent evictions do not generate any coherence traffic. Thus, they do not update the sharing information at the directory, i.e., the sharing code codifies a superset of the actual sharers. This also increases the number of invalidation messages.

Figure 6 shows the number of write misses, normalized with respect to *BitVector*, classified by the number of nodes (color-coded) that receive invalidation messages, i.e., the number of sharers assumed by the directory. The five bars for each application in the figure represent, respectively, the *BitVector*, the *OnePointer*, the *TwoPointers*, the *SingleList* and the *DoubleList* protocol. While the total amount of misses remains almost constant, as expected, the protocols that keep the set of sharers with less accuracy will need to send invalidations to more nodes per miss.

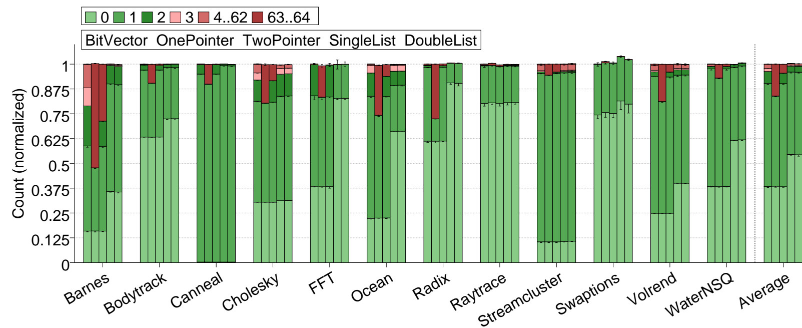
We can observe two general trends. First, the average number of contacted nodes is very low. In fact, few misses need more than one invalidation even in the *OnePointer* protocol. Second, as expected, the protocols that require notification of replacements for shared blocks, i.e., the list-based protocols, contact a lower number of nodes on each write miss.

In particular, the protocol with a single pointer in the L2 cache (*One-Pointer*) will require a broadcast for about one out of ten write misses, on average, when 16 cores are assumed. This fraction increases up to 14% for 64-core CMPs, where also the penalty of broadcasts is, at least, four times greater, since it is sent to 64 nodes instead of to 16. In some cases (e.g., *Ocean*) about half of the misses require a broadcast. This factor limits the scalability of the *OnePointer* protocol and it will be reflected in both the achieved execution time and energy consumption.

The *TwoPointers* protocol considerably reduces the number of broadcasts required to invalidate the sharers to just 3% for 16-core CMPs and 4% for 64-core CMPs. However, both the frequency and the penalty of broadcast messages still increase with the number of cores.



(a) 16 cores



(b) 64 cores

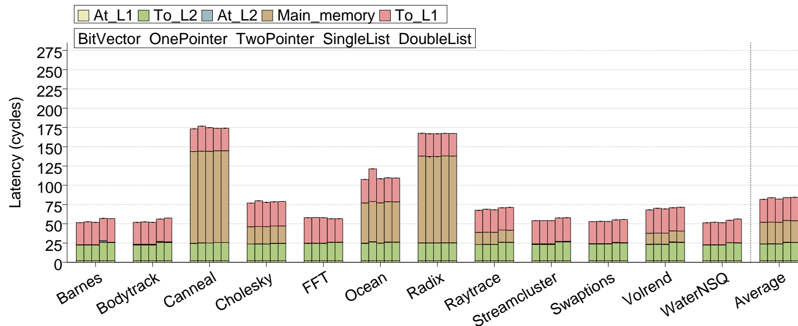
Fig. 6: Write misses classified by the number of nodes invalidated.

The *SingleList* and *DoubleList* protocols seldom send two or more notifications on average for a write miss. Therefore, although invalidation messages are generated sequentially as the list of sharers is traversed, the average latency of write misses is not expected to increase too much with respect to *BitVector*, as will be shown next.

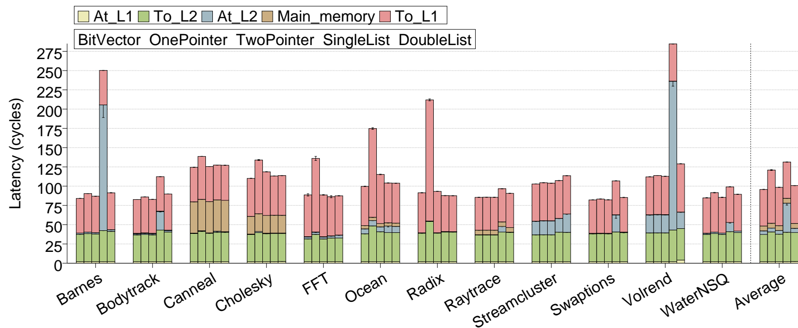
6.2 Latency of L1 cache misses

The latency of L1 cache misses is a key aspect of the performance of a multicore architecture since it determines the average memory latencies perceived by processor cores. The sharing code employed by the coherence directory can affect latency significantly, specially as the number of cores in the chip increases.

For the accesses that miss in the L1 cache, Figure 7 plots the observed average latencies for both a 16-core CMP and a 64-core CMP. Each bar in this figure has been split in five parts: the time spent in accessing the L1 cache (*At_L1*) and also as a consequence of stalls due to on-going coherence actions



(a) 16 cores



(b) 64 cores

Fig. 7: L1 cache miss latency.

or exhausted MSHR capacity; the time from the L1 to the L2 (*To_L2*) required to access the directory information; the time spent waiting until the L2 can attend the miss (*At_L2*), mostly due to on-going transactions on the same memory block; the time spent waiting to receive the data from main memory (*Main_memory*) in case that the requested block is not present in any cache in the chip; and the time from when the L2 sends the data or forwards the request until the requester receives the memory block and can resolve the miss (*To_L1*). The *Main_memory* time will be zero for a large number of L1 cache misses because the data can be found on chip most times. However, it still represents a significant part of the average miss latency, mostly for configurations with a lower number of cores.

In a CMP comprised of 16 cores, latency of L1 cache misses is not much affected by the sharing code employed by the coherence directory (Figure 7(a)). There is only a small increase in the *To_L1* time for the *OnePointer* protocol which can be better appreciated in *Ocean*. This is due to an increase in the latency of write misses, which require broadcasting invalidation messages for about 28% of write misses, as previously shown in Figure 6(a). Although

Barnes also shows an elevated fraction of broadcasts per write (36%), the low frequency of write misses lessen the effect in the overall access latency.

In contrast, the results for the 64-core CMP configuration show more differences in the latency of L1 cache misses depending on the sharing code organization (Figure 7(b)). There are two main reasons for such a variation. The first reason is the increase in the *To_L1* latency in the *OnePointer* protocol, due to the higher number of cores that will receive invalidation messages and will have to answer with an acknowledgement message. This can cause serious bottlenecks especially when multicast or broadcast support is not provided at NoC (Network-on-Chip) level. The second reason is the sharp increase observed in many benchmarks of the time spent waiting for the L2 cache to attend the miss (*At_L2*) in the *SingleList* protocol.

We can appreciate the increase in the latency of this part of the process when going from 16 to 64 processors even for the *BitVector* protocol, where some applications already start suffering the effects of L2 contention. However, the *SingleList* protocol exacerbates this problem. This happens because the L2 is blocked much more often in this protocol.

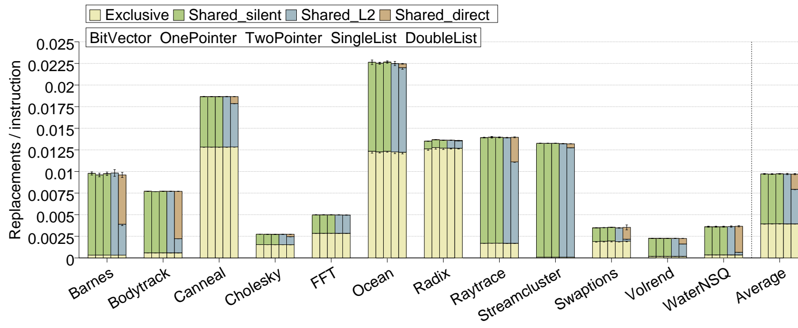
In effect, *SingleList* keeps the L2 blocked during updates to the list of sharers to ensure mutual exclusion of list updates and avoid inconsistencies in the list, (i.e., only one insertion or one deletion can be done at the same time to the same list). This forces the L2 cache to remain blocked and unable to answer to other requests to the same memory block. Replacements of shared blocks, as explained in Section 3, require contacting sequentially half the sharers of a block on average, keeping the block locked in L2 during all that time. Differently, in the *BitVector* protocol, the L2 does not need to intervene at all in replacements of shared blocks since they are silent. For this reason, L2 contention in *SingleList* will increase with the number of cores accessing the block.

Employing a doubly-linked list (*DoubleList*) makes possible to avoid blocking the L2 during replacements. In *DoubleList*, not all updates to the list are performed in mutual exclusion and the L2 does not need to intervene in many replacements of shared blocks, as explained in Section 3. This way, the time spent at L2 by this protocol is very similar to *BitVector*, although the time spent accessing the L1 (*At_L1*) is increased slightly because the L1s have to be partially locked while a following node is performing a replacement.

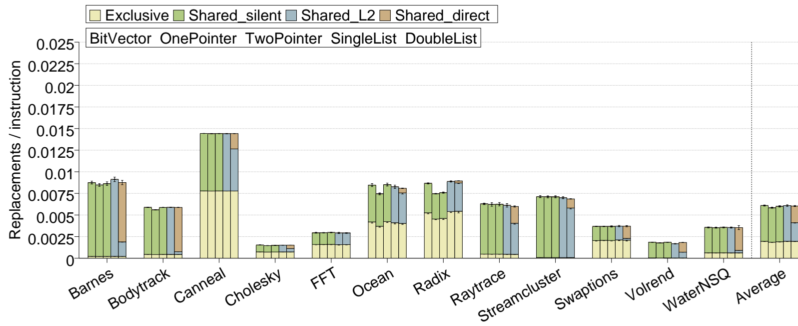
Finally, despite the serial nature of the invalidations in the list-based protocols, the latency of the *To_L1* fraction does not increase. This is due to the very low frequency of writes to memory blocks with a large number of sharers, as shown in the previous section.

6.3 L1 cache replacements

L1 cache replacements play an important role in the efficiency of the cache coherence protocol because they may require updating the sharing code. This section analyzes four different types of replacements that can happen depend-



(a) 16 cores



(b) 64 cores

Fig. 8: L1 cache replacements typology.

ing on the protocol and the status of the block. Figure 8 plots the number of replacements per instruction executed in each of the following categories: *Exclusive*, *Shared_silent*, *Shared_L2*, and *Shared_direct*.

In all the evaluated protocols, when *exclusive* blocks are evicted the sharing code is updated in to reflect that there are not copies of the block in any private cache. The transaction required to perform this update is exactly the same in all protocols. Once the replacement is completed, there is no need of keeping an entry in the directory structure, or in case of having the directory along with the L2 cache entries (as we assume in this paper), there is no need of sending invalidation messages if the block is subsequently expelled from the L2 cache. The number of exclusive evictions does not significantly change across the protocols.

On the other hand, replacement of *shared* blocks proceeds differently depending on the protocol. Figure 8 shows that the fraction of *shared* replacements is, on average, larger than the fraction of *exclusive* replacements, and that this fraction increases with the number of cores.

As discussed in Section 6.1, protocols that use centralized sharing codes usually implement *silent* replacements, which do not generate any coherence

transactions, although extra invalidation messages may be produced later upon write misses or L2 cache replacements.

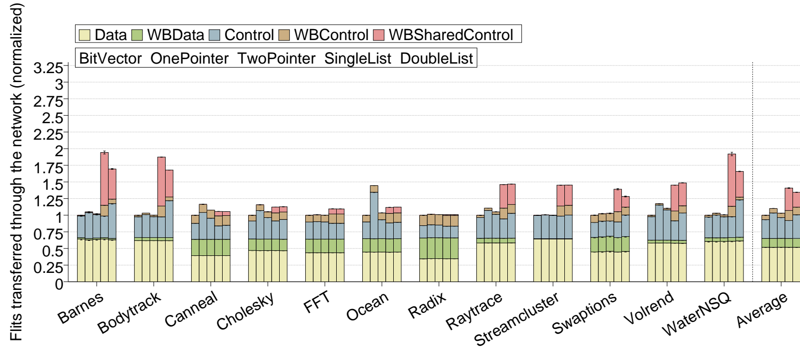
In protocols that employ sharing codes distributed across the L1 caches, such as *SingleList* and *DoubleList*, replacements in the L1 cache imply loss of the sharing information kept in that cache, which would render the list of sharers broken due to the now missing link. Hence, these protocols need to update the sharing status of memory blocks on each replacement. These updates have two main consequences. First, updating the coherence information of a particular block requires mutual exclusion in the case of *SingleList*, which affects the waiting time at the L2 (*At_L2*), as already shown in Figure 7. Second, it increases the traffic in the interconnect, as it will be shown in the next section.

The advantage of the *DoubleList* protocol with respect to the *SingleList* protocol is that the updates performed by replacements in *DoubleList* can often be done without accessing the L2 cache, and therefore without blocking other requests that may need updating the list of sharers. We name these replacements as *Shared_direct*, while the replacements that require to block the L2 entry are called *Shared_L2*. In *DoubleList*, the L2 is involved during a shared replacement only when the replacing node is the first of the list of sharers, and then only for a shorter period of time than in the case of *SingleList* because the list of sharers does not need to be traversed. We can observe that, on average, the fraction of replacements that can avoid blocking the L2 in *DoubleList* increases with the number of nodes in the system. For a 64-core CMP configuration this fraction is almost half of the total number of shared replacements, on average. This is why *DoubleList* reduces the *At_L2* time, and what enables this protocol to offer higher scalability than *SingleList* in terms of performance.

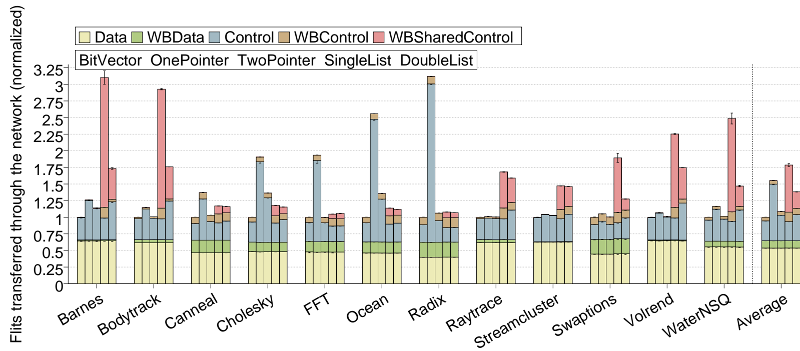
6.4 Network traffic

Figure 9 shows the traffic measured in flits and normalized with respect to *BitVector* that travels through the network for configurations of 16 and 64 cores with each protocol. Traffic has been divided in the following categories: data messages due to cache misses (*Data*), data messages due to replacements (*WBData*), control messages due to cache misses (*Control*), control messages due to replacements of private data (*WBControl*) and control messages due to replacements of shared data (*WBSharedControl*).

As it can be seen in the results for the 16-core configuration (Figure 9(a)), *OnePointer* results into increased traffic due to control messages (*Control*) with respect to *BitVector*. This is because this protocol resorts on broadcasting invalidation messages whenever there is more than one sharer. *TwoPointers* also increases this traffic category but to a much lesser extent, since broadcasts are only performed when there are more than two sharers, which is infrequent as shown in 6.1.



(a) 16 processors



(b) 64 processors

Fig. 9: Interconnection network traffic.

On the other hand, *SingleList* has slightly less traffic due to control messages for misses than *BitVector*. This is because write misses generate fewer messages in *SingleList*. The reasons for this are twofold. First of all, silent replacements for shared data are not possible in *SingleList*, and therefore, only actual sharers receive invalidation messages. And second, only the last invalidated sharer sends an acknowledgement message to the requester in *SingleList*, while every invalidated sharer needs to send an acknowledgement in the case of *BitVector* (and also in *OnePointer* and *TwoPointers*). However, *SingleList* increases significantly the traffic due to replacements, especially in the case of the replacements of shared data (WBSharedControl) which are done silently in the case of the protocols using centralized sharing codes. As previously shown, the replacement process, which updates the sharing list sequentially, contributes also to increasing contention at the L2 cache level, which will negatively affect performance.

DoubleList traffic due to control messages is higher than that of *BitVector*. Although *DoubleList* needs only one acknowledgement for each write miss

like *SingleList* and silent replacements for shared data are neither possible in *DoubleList*, it entails additional messages for each read miss to a block that already has at least one sharer because it needs to update the previous sharer pointer of the former first sharer of the block, as explained in Section 3.2.1.

For the 64-core configuration (Figure 9(b)), we can differentiate two groups of benchmarks: those where the *Control* traffic introduced by the broadcast in *OnePointer* dominates and those where the *WBSharedControl* traffic introduced by the shared replacements in *SingleList* dominates. Within the first category would be *Canneal*, *Cholesky*, *FFT*, *Ocean*, and *Radix*. Whereas *Barnes*, *Bodytrack*, *Raytrace*, *Streamcluster*, *Swaptions*, *Volrend*, and *WaterNSQ* fall into the second category.

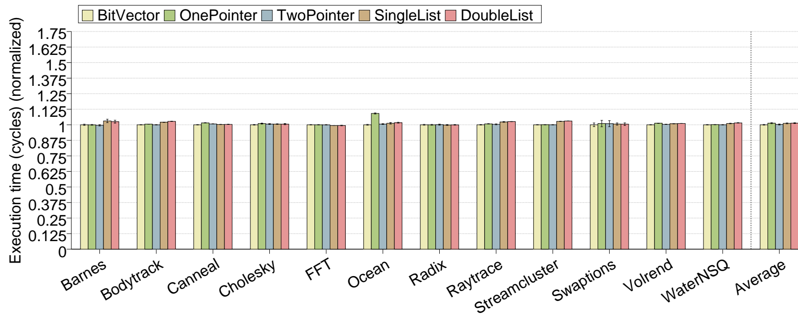
Both *OnePointer* and *SingleList* show poor scalability with respect to interconnection traffic. While in *OnePointer* the cost of invalidation broadcasts grows quickly with the number of cores, the cost of replacements in *SingleList* so it does because the lists of sharers that have to be traversed tend to be longer. *TwoPointers* and *DoubleList* are able to reduce the respective problems that *OnePointer* and *SingleList* introduce at the expense of requiring more area. *TwoPointers* reduces traffic by reducing the number of times that broadcasting invalidations occurs, and *DoubleList* reduces the number of messages per replacement and, more importantly, makes that number a constant (3) instead of depending on the number of sharers of the list.

6.5 Execution time

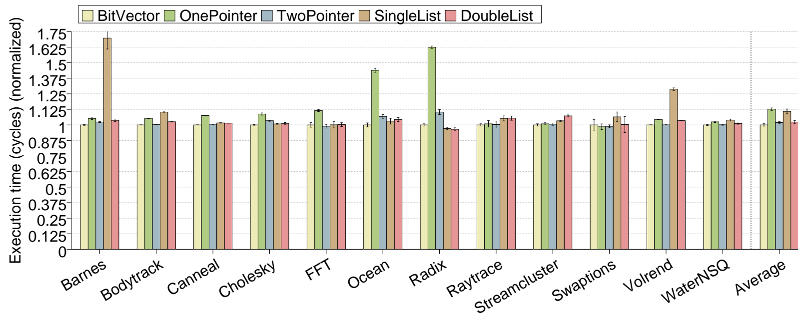
After analyzing separately the aspects of the applications that may affect their execution time depending on the protocol employed, this section looks at the resulting execution time in each case. As in the previous figures, Figure 10 shows the execution times normalized with respect to *BitVector*, for both 16- and 64-core configurations.

The election of the coherence directory barely affects execution time for the 16-core configuration (Figure 10(a)). Only for *Ocean*, a noticeable performance degradation is observed for *OnePointer* (about 10%). We have observed that this is due to the increase in the *To_L1* latency reported in Figure 7(a)).

However, when 64 cores come into play (Figure 10(b)) some applications suffer a significant increase in their execution time especially for the *OnePointer* and *SingleList* protocols. This increase can be observed most clearly in *Ocean* and *Radix* for *OnePointer* and in *Barnes* and *Volrend* for *SingleList*. If we look back to the miss latency results (Figure 7(b)), we can see that, in the case of *SingleList*, the most affected applications are those whose *At_L2* time was most increased due to the extra L2 contention created by shared replacements. On the other hand, the most affected applications with *OnePointer* are those whose *At_L2* time increases the most. These are, precisely, those that require broadcast invalidations most often, as can be seen if we look back at figure 6(b).



(a) 16 cores



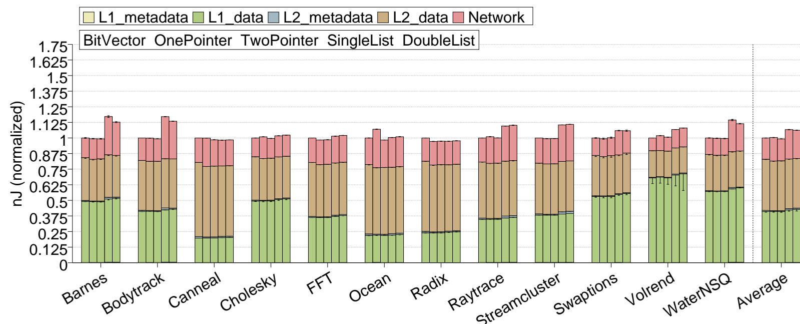
(b) 64 cores

Fig. 10: Execution time.

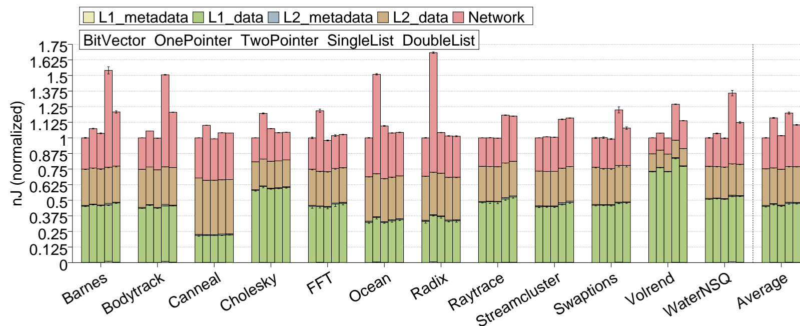
6.6 Energy consumption

Finally, we show the energy consumption (both dynamic and static) for the different protocols evaluated. The dynamic energy consumption normalized with respect to *BitVector* is shown in Figure 11. We have considered the consumption of the L1 caches (both data and metadata), the consumption of the L2 caches (both data and metadata), and the consumption of the interconnection network. For the 16-core configuration, *OnePointer*, *TwoPointers*, and *DoubleList* consume less dynamic energy than our baseline, on average. This is mostly due to the reduction in the number of accesses to the L2 cache. In contrast, for the 64-core configuration, the average consumption increases in *OnePointer*, *SingleList*, and *DoubleList* due to the increase in network traffic that these protocol entail. Although in *DoubleList* the number of accesses to the L2 cache is considerably reduced, the extra traffic generated by the *WB-SharedControl* messages (see Figure 9(b)), makes it consume more dynamic energy.

The static (or leakage) energy is shown in Figure 12, again normalized to *BitVector*. We account for the consumption of the L1 and L2 caches during the



(a) 16 cores



(b) 64 cores

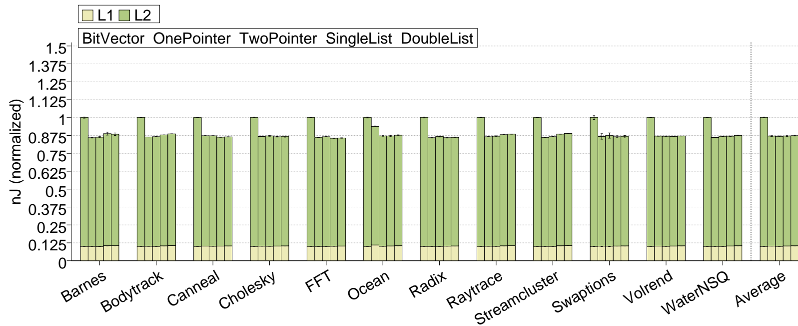
Fig. 11: Dynamic energy consumption.

execution of the applications. For the 16-core configuration, a clear reduction in the leakage due to replacing the bit-vector with a pointer in the L2 cache can be appreciated. This represents an average energy reduction of 12.5%.

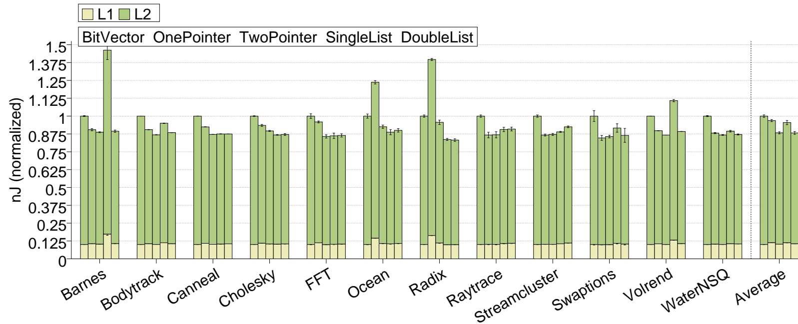
For the 64-core configuration we can observe an increase in the leakage due to the increase in the execution time of some applications for the *OnePointer* and *SingleList* protocols. Still, this increase pays off the reduction in the required area to keep a bit-vector sharing code.

7 Conclusions

Although cache coherence protocols based on the use of a distributed sharing code (mainly a doubly-linked list) were popular in the context of shared-memory multiprocessors during the 90s, they have gone completely unnoticed in the multicore era. In this work, for the first time, we have performed a comprehensive evaluation of two coherence directory implementations employing two distributed sharing codes (*SingleList* and *DoubleList*) in the context of future manycore architectures. *SingleList* is based on the use of simply-



(a) 16 cores



(b) 64 cores

Fig. 12: Energy leakage.

linked lists, thus being the lowest-overhead coherence directory that can be designed with a distributed sharing code. On the other hand, *DoubleList* employs doubly-linked lists and resembles the coherence protocols based on distributed sharing codes implemented in several commodity multiprocessors.

We have presented detailed results in terms of performance, memory overhead, network traffic and energy consumption, and we have compared these two distributed sharing codes against three well-known centralized ones. Simulation results of show that all the configurations obtain similar results in terms of performance and energy consumption when the number of cores is not so big (16 cores). However, important differences between the alternatives appear when the number of cores is increased to 64, underscoring the significant incidence that the coherence directory may have in future manycores. In particular, for large core counts, we have found that *SingleList* performance is worse than the base case (*BitVector*) because of the increase in traffic and L2 contention due to L1 replacements. We also observe that *DoubleList* reduces leakage energy (approximately 13% on average) and reaches the performance levels of the non-scalable memory-consuming bit-vector directory with just a

small increase in dynamic energy consumption (about 10% on average). The latter comes from the additional traffic that *DoubleList* entails as a consequence of having non-silent replacements for shared data.

For 64 cores or less, the *TwoPointers* protocol could be a good option despite needing almost twice as much memory as *DoubleList*. However, it is important to note that the *TwoPointers* protocol resorts to broadcasting coherence messages when the number of sharers is greater than 2, which makes it non-scalable because, for larger core counts, performing broadcasts for invalidations and processing all the responses, even if very infrequently, would surely saturate the interconnection network, the home directory and the requestor.

We think that based on the reported results we can conclude that coherence directories based on distributed sharing codes are an appealing solution to the scalability problem of the coherence directory in future manycore architectures, and that it is worth exploring solutions to the increased overhead that replacements of shared data have in these protocols. In fact, this is something that we plan to tackle as part of our future work.

Acknowledgements This work has been supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant "TIN2012-38341-C04-03" and by the Fundación Séneca-Agencia de Ciencia y Tecnología de la Región de Murcia under grant "19295/PI/14".

References

1. J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, B. Sinharoy, POWER4 system microarchitecture, IBM Journal of Research and Development 46 (1) (2002) 5–25.
2. D. Kanter, Knights landing details, http://www.realworldtech.com/knights-landing-details/ (Jan. 2014).
3. M. Mattina, Architecture and performance of the tilera TILE-Gx8072 manycore processor, Invited presentation at 21st HotInterconnects Symp. (2013).
4. S. Borkar, Thousand core chips: a technology perspective, in: 44th Design Automation Conference (DAC), 2007, pp. 746–749.
5. M. M. K. Martin, M. D. Hill, D. J. Sorin, Why on-chip cache coherence is here to stay, Communications of the ACM 55 (7) (2012) 78–89.
6. M. E. Acacio, J. González, J. M. García, J. Duato, A two-level directory architecture for highly scalable cc-NUMA multiprocessors, IEEE Transactions on Parallel and Distributed Systems (TPDS) 16 (1) (2005) 67–79.
7. J. M. Owen, M. D. Hummel, D. R. Meyer, J. B. Keller, System and method of maintaining coherency in a distributed communication system, U.S. Patent 7069361 (Jun. 2006).
8. M. Ferdman, P. Lotfi-Kamran, K. Balet, B. Falsafi, Cuckoo directory: A scalable directory for many-core systems, in: 17th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2011, pp. 169–180.
9. D. E. Culler, J. P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, Inc., 1999.
10. D. B. Gustavson, The scalable coherent interface and related standards proyects, IEEE Micro 12 (1) (1992) 10–22.
11. R. Clark, K. Alnes, An SCI interconnect chipset and adapter, in: HotInterconnects Symp. IV, 1996, pp. 221–235.
12. T. Lovett, R. Clapp, STiNG: A cc-NUMA computer system for the commercial marketplace, in: 23rd Int'l Symp. on Computer Architecture (ISCA), 1996, pp. 308–317.

13. R. Thekkath, A. P. Singh, J. P. Singh, S. John, J. L. Hennessy, An evaluation of a commercial cc-NUMA architecture: The CONVEX Exemplar SPP1200, in: 11th Int'l Symp. on Parallel Processing (IPPS), 1997, pp. 8–17.
14. R. Fernández-Pascual, A. Ros, M. E. Acacio, Characterization of a list-based directory cache coherence protocol for manycore cmps, in: 3rd Workshop on On-chip Memory Hierarchies and Interconnects (OMHI 2014), 2014, pp. 254–265.
15. M. Thapar, B. Delagi, Stanford distributed-directory protocol, Computer 23 (6) (1990) 78–80.
16. D. James, A. Laundrie, S. Gjessing, G. Sohi, Scalable coherent interface, Computer 23 (6) (1990) 74–77.
17. L. M. Censier, P. Feautrier, A new solution to coherence problems in multicache systems, IEEE Transactions on Computers (TC) 27 (12) (1978) 1112–1118.
18. D. Chaiken, J. Kubiatowicz, A. Agarwal, LimitLESS directories: A scalable cache coherence scheme, in: 4th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS), 1991, pp. 224–234.
19. M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, The raw microprocessor: A computational fabric for software circuits and general purpose programs, IEEE Micro 22 (2) (2002) 25–35.
20. P. Hammarlund, A. J. Martínez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, T. Burton, Haswell: The fourth-generation intel core processor, IEEE Micro 34 (2) (2014) 6–20.
21. R. Kalla, B. Sinharoy, W. J. Starke, M. Floyd, POWER7: IBMs next-generation server processor, IEEE Micro 30 (2) (2010) 7–15.
22. M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Saha, D. Sheahan, L. Spracklen, A. Wynn, UltraSPARC T2: A highly-threaded, power-efficient, SPARC SoC, in: IEEE Asian Solid-State Circuits Conference, 2007, pp. 22–25.
23. M. Zhang, K. Asanović, Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors, in: 32nd Int'l Symp. on Computer Architecture (ISCA), 2005, pp. 336–345.
24. A. Ros, M. E. Acacio, J. M. García, Scalable directory organization for tiled CMP architectures, in: Int'l Conference on Computer Design (CDES), 2008, pp. 112–118.
25. A. Ros, M. E. Acacio, J. M. García, A scalable organization for distributed directories, Journal of Systems Architecture (JSA) 56 (2-3) (2010) 77–87.
26. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in: 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2005, pp. 190–200.
27. M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, Computer Architecture News 33 (4) (2005) 92–99.
28. M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, P. Faraboschi, How to simulate 1000 cores, Computer Architecture News 37 (2) (2009) 10–19.
29. V. Puente, J. A. Gregorio, R. Beivide, SICOSYS: An integrated framework for studying interconnection network in multiprocessor systems, in: 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, 2002, pp. 15–22.
30. S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, N. P. Jouppi, Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures, in: 42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO), 2009, pp. 469–480.
31. A. Banerjee, P. T. Wolkotte, R. D. Mullins, S. W. Moore, G. J. M. Smit, An energy and performance exploration of network-on-chip architectures, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 17 (3) (2009) 319–329.
32. P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, B. Hughes, Blade computing with the AMD Opteron™ processor ("Magny Cours"), in: 21st HotChips Symp., 2009.

33. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, in: 22nd Int'l Symp. on Computer Architecture (ISCA), 1995, pp. 24–36.
34. C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: 17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), 2008, pp. 72–81.
35. A. R. Alameldeen, D. A. Wood, Variability in architectural simulations of multi-threaded workloads, in: 9th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2003, pp. 7–18.