Optimization of a Linked Cache Coherence Protocol for Scalable Manycore Coherence*

Ricardo Fernández-Pascual, Alberto Ros, and Manuel E. Acacio

Dpto. de Ingeniería y Tecnología de Computadores.
Universidad de Murcia (SPAIN)
{rfernandez,aros,meacacio}@ditec.um.es

Abstract. Despite having been quite popular during the 1990s because of their important advantages, linked cache coherence protocols have gone completely unnoticed in the multicore wave. In this work we bring them in the spotlight, demonstrating that they are a good alternative to other solutions being proposed nowadays. In particular, we consider in this work the case for a simply-linked list-based cache coherence protocol and propose two techniques, namely Concurrent Replacements (CR) and Opportunistic Replacements (OR), aimed at palliating the negative effects of replacements of clean data. Through detailed simulations of several SPLASH-2 and PARSEC applications, we demonstrate that, armed with CR and OR, simply-linked list-based protocols are able to offer the performance of a non-scalable bit-vector directory at the same time that scalability to larger core counts is preserved.

Keywords: Manycores, Cache coherence, Exact sharer enconding, Scalability, Singly-linked list, Area overhead, Execution time, Network traffic

1 Introduction

As technology allows the fabrication of chip-multiprocessors with dozens or even hundreds of cores, the organization of the coherence directory responsible for keeping track of the sharers of cached memory blocks becomes a first-class design concern. Ideally, a coherence directory should satisfy three basic requirements [15]: *i)* small area, energy, and latency overheads that scale well with the number of cores; *ii)* exact sharer representation to minimize resulting coherence traffic; and *iii)* avoidance of directory-induced invalidations. The latter can arise with sparse directories due to their limited capacity and associativity, and could be reduced by doing a more efficient use of their entries [5, 8, 7]. However, ensuring exact sharer representation is usually hard, since it entails increased area and energy requirements (if bit vectors are used), directory-induced invalidations (for limited pointers, when a pointer recycling overflow strategy is employed),

* This work has been supported by the Spanish MINECO, as well as European Commission FEDER funds, under grants "TIN2012-38341-C04-03" and "TIN2015-66972-C5-3-R", and by the Fundación Séneca-Agencia de Ciencia y Tecnología de la Región de Murcia under grant "19295/PI/14".

or extra directory latency and/or non-conventional structures (e.g., SCD [15] requires a non-conventional ZCache architecture).

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 [6]: those that store the sharing information about all the cached copies of each block in a single and fixed place, the home node of that block (we call them directories with *centralized* sharing codes), and those in which sharing information is distributed among the caches holding copies of each block and the home node, which only contains a pointer to one of the sharers (we call them directories with *distributed* sharing codes). Surprisingly, all recent proposals have concentrated on the first type of directories, and despite having been popular in the context of shared-memory multiprocessors during the 1990s [10, 11, 17], directories with distributed sharing codes have gone completely unnoticed in the multicore era.

Directories based on distributed sharing codes have several important advantages over their centralized counterparts. First of all, they ensure scalability since they employ pointers whose size increases logarithmically with the number of cores. Fig. 1 shows that the amount of memory required by the simplest directory able to provide exact sharer representation based on a distributed sharing code (that using a simply-linked list, *List*) remains below 2% as the number of cores is increased to 1024. The memory requirements of the recently proposed SCD [15] and two directories with centralized sharing codes (namely, *BitVector* and *1-Pointer*) are also shown for comparison purposes. *BitVector* does not scale because the number of bits per directory entry increases linearly with the number of cores, but it is able to offer the best performance. In [15], SCD has been shown to approximate the performance of *BitVector* but its area requirements do not scale as well as *List*. Finally, *1-Pointer* incurs an area penalty similar to *List* but it does not guarantee exact sharer representation, which compromises performance.

A second advantage of directories based on distributed sharing codes is that they naturally lead to more efficient use of the precious directory information, dynamically devoting more resources to those memory blocks with more sharers and less to those that do not need them (every cached data has at least one pointer assigned, which is the minimum to have its precise location, and at most as many pointers as sharers). And, finally, a third one is that conversely to other proposals based on the use of multi-hashing [15], directories based on distributed sharing codes do not require non-conventional cache structures. However, some disadvantages have been reported also, such as increased latency of write misses due to a sequential invalidation of sharers, high-latency replacements of shared blocks since the list structure must be preserved, and the necessity of modifying private caches to include pointers.

In our previous work [9], we brought a singly-linked list-based directory similar to that described in [16] (*List*) to the multicore world. We found out that *List* was specially appealing since it entailed minimal additions to the critical

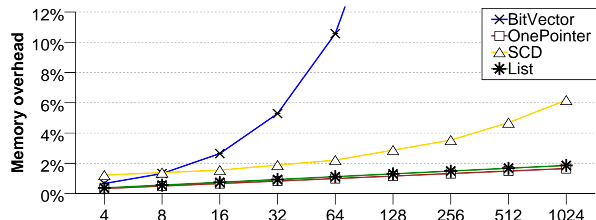

Fig. 1. Memory overhead as the number of cores increases.

private L1 caches (just one pointer per cache entry) and at the same time it was able to ensure scalability in terms of directory memory overhead (as shown in Figure 1). However, we noticed important performance degradations (almost 20% on average) when compared with *BitVector*. Particulary, we discovered that the most important source of inefficiency in *List* was due to the extra messages necessary to handle clean shared data replacements. Now, we propose in this work two novel techniques that minimize the negative impact that replacements of clean data in a simply-linked protocol have on performance: *Opportunistic replacements* (OR) and *Concurrent replacements* (CR). OR tries to mix pending replacements of clean data at the private caches with an on-going replacement (all referred to the same address), thus minimizing the number of replacements that the home node must manage. CR allows the home node to deal with load misses to a particular address whilst a replacement to the same address is progressing.

To the best of our knowledge, this is the first work showing how in manycores a directory with a distributed sharing code (*List*) can reach the performance of the most efficient but unfeasible (non-scalable) directory with a centralized bit-vector sharing code (*BitVector*) at the same time that scalability to larger core counts is guaranteed. The only overhead that we have observed in *List* is some increase in terms of average network traffic but it has no impact on final performance.

2 *List*: A simply-linked list protocol

The *List* protocol analyzed in this work has been developed by modifying the base MESI protocol. Although different, *List* resembles previous proposals based on singly-linked lists like [16]. The complexity of the resulting protocol is very close to that of the base protocol from which it has been derived.

Distributed sharing codes store directory information for each memory block in a distributed way, instead of keeping it centralized in the home node. In *List*, the home node stores the identity of only one of the possibly many sharers of the memory block while the rest of sharers are represented using a linked list constructed through one pointer in each of the L1 cache entries[1]. Storing the sharing information this way implies that, to update it, some messages need to be

[1] Without generality loss, we assume private L1 caches in each core and an inclusive, shared L2 cache distributed between them.

exchanged between the sharers and the home node which would not be needed in a protocol using a centralized sharing code, but it achieves a much higher scalability in terms of memory overhead which results in lower area requirements and static energy consumption.

Whereas the amount of memory required per directory entry with a centralized bit-vector sharing code grows linearly with the number of processing cores (i.e., one bit per core), it grows logarithmically for distributed sharing codes (Fig. 1). Although distributed sharing codes need some additional information in each L1 entry (pointers), this is not a problem for scalability because the number of entries in the private caches is always much smaller than in the shared cache banks. Figure 1 shows 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 for different numbers of cores, assuming 4-way L1 caches with 128 sets and 16-way L2 caches with 256 sets per core and 64 byte blocks.

The starting point *List* protocol considered in this work is a slightly improved version of the one evaluated in [9]. The current version has one message less in the critical path of replacements (but still has the same total number of messages).

List stores the sharing information of a block in a distributed manner. The home L2 node keeps a pointer to the first sharer and each sharer keeps a pointer to another sharer, creating a linked list of sharers. The last sharer keeps a null pointer, which is codified as the identity of the sharer itself so that no additional bits are needed (i.e., the last node points to itself).

In *List*, without optimizations, updates to the list of sharers are serialized by the home node, which remains blocked (i.e., other requests for this block are not attended) until the modification of the list has been completed. This avoids races that could corrupt the list.

For requests to non-cached or private blocks, *List* behaves almost identically to a centralized protocol. The home L2 bank, after sending the data and receiving an *Unblock* message from the requester, stores the identity of the new and only sharer in the block's pointer. The differences come with read and write misses and replacements of shared blocks.

For read misses to blocks with at least one sharer, the new sharer is inserted at the beginning of the list (i.e., the new sharer will point to the previous first sharer and the L2 will point to the new sharer). The identity of the previous first sharer is sent to the requester along with the data, which can be sent by the L2 (for the S state) or by the previous first sharer after it receives the forwarded request (for the M or E state). No additional messages are required with respect to a centralized protocol in either case.

For write misses, the invalidation of all sharers is done in parallel to sending the data to the requester. But, while in a centralized protocol the home node can perform the invalidations in parallel by sending them directly to every sharer (because it has the complete sharing information), in *List* the L2 sends the invalidation only to the first sharer, which forwards it to the next sharer and so on until arriving to the last one, which sends an acknowledgment to the requester. This means that latency may increase specially for long lists of sharers

because the invalidations are processed sequentially. On the other hand, while in a centralized protocol every sharer needs to send an acknowledgment to the requester, in *List* only the last sharer sends it, hence the number of messages used to resolve a write miss is the same or less than with a centralized protocol.

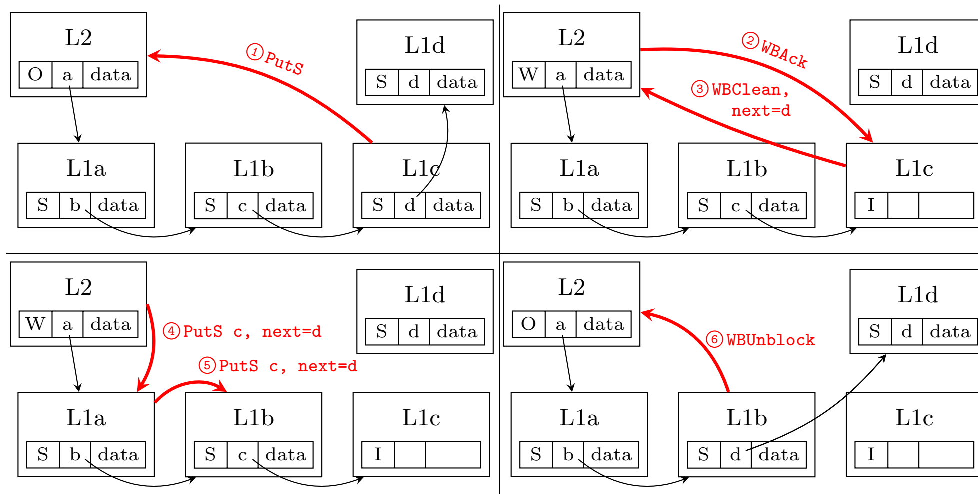

Fig. 2. Example of shared block replacement in *List*

Replacements of blocks shared by only one node are handled exactly the same as in a centralized protocol. However, the most determinant difference for performance is the way that replacements of shared blocks are done. In a centralized protocol, these replacements can happen silently (i.e., the sharer discards the data and does not inform the home node about it) while *List* needs to involve the home node and other sharers in a potentially long process. This is so because the sharing information (i.e., the pointer to the next sharer) is stored alongside the data, and discarding it would leave the list of sharers unconnected. Instead, as shown in figure 2, before a shared memory block can be replaced, a replacement request (①) is sent to the L2. When the L2 receives it and it is ready to handle it, it responds with a message authorizing the replacement (②). This message is answered by the replacing node with another one (③) that carries the value of the pointer kept at this L1, which indicates the *next sharer* in the list. After sending it, the L1 cache can discard the pointer from its cache and it will not be contacted again regarding this transaction. Upon receiving the message with the *next sharer*, the L2 needs to update the sharing list (which at this point is momentarily disconnected because the replacing node has discarded its pointer). For this, if the replacing node coincides with the sharer stored at the L2 (i.e., it is the first sharer of the list), then the value of the pointer at the L2 is changed to point to the *next sharer*. Otherwise, the L2 cache forwards the replacement request to the first sharer (④) and the message keeps propagating through the list of sharers until the node that preceded the

replacing node in the list is reached (⑤). This is identified because its pointer to the next sharer will match the requester identity. At this point, the preceding node updates its pointer with the information included in the message (the *next sharer*), reconnecting the list of sharers. After updating its next pointer, the node sends an acknowledgment (⑥) to the L2 and the operation completes. Notice that the identity of the *next sharer* 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.

The fact that replacements for shared data in the *List* 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 (from other nodes) to the replaced address because those misses cannot be attended by the L2 until the replacement has finished.

3 Optimizing shared replacements in *List*

The main reason for the execution time and traffic difference between *BitVector* and *List* is due to the replacement of shared blocks [9]. To mitigate this problem, we have designed two optimizations to *List* which can be used in combination or independently of each other: *Opportunistic replacements*, which mainly affect the behavior of L1 controllers, and *concurrent replacements*, which affect only the behavior of the L2 controller.

It is important to understand that shared replacements do not increase the latency of the miss that actually caused the replacement because all protocols considered in this work assume that the L1 caches have writeback buffers[2]. Instead, the replacement affects the latency of subsequent misses to the same or different addresses due to two reasons: the additional traffic, which can be absorbed by the network easily, and, more importantly, the additional time that the L2 controller needs to be blocked while handling the replacement.

3.1 Opportunistic replacements (OR)

We noticed that, in some applications, several sharers of the same block frequently will request to replace it almost simultaneously. When this happens in *List*, the L2 has to process all these requests sequentially. Each replacement is a

[2] In the case of clean shared replacements, the writeback buffer only needs to store the sharing information, not the data. Due to its very small size in *List*, this information may alternatively be kept in a miss status holding register (MSHR) or similar structure.

potentially long process because the list of sharers needs to be traversed (half of it, on average), and the L2 has to be blocked during the process, resulting in delaying other misses. *Opportunistic replacements* take advantage of the traversal of the list required for a shared replacement request to perform the replacement of other nodes that have also requested it. This way, fewer traversals are needed.

It works as follows. After an L1 requests a replacement, it keeps waiting until it receives permission from the L2, as previously described. If the L1 sees a message for a replacement requested by another L1 while it is waiting, instead of forwarding it to its next sharer it will send to the node who sent it (which may be either its previous sharer or the L2) an opportunistic replacement request including in the same message the identity of its next sharer and then it will discard the sharing information (the data had been discarded already), leaving the list temporarily unconnected.

Upon reception of the opportunistic replacement request message, the previous sharer will update its pointer to the next sharer, reconnecting the list, and then forward the original replacement request again to its new next sharer (or finish the transaction if the new next sharer happens to be the requester of the original replacement).

At this point, the L1 waiting to replace has been already disconnected from the list and when it eventually receives the authorization to replace from L2, it will answer with a *nack*, quickly unblocking the L2 without needing to traverse the list again. Notice that unblocking the L1 directly after it gets disconnected from the sharers list would lead to a protocol race and either deadlock or an incorrect list update because the L1 had already sent a replacement request to the L2 that it will eventually receive and process expecting some response.

This way, *Opportunistic replacements* reduce both the time that the L2 remains blocked and the traffic due to the replacements by means of avoiding repeated traversals of the list of sharers.

3.2 Concurrent replacements (CR)

As stated before, the effect of shared replacements in the execution time of *List* is mainly due not to the increase in traffic neither to the increase in the time that L1 nodes spend performing replacements. Rather, what impacts performance the most is the time that the L2 stays blocked during the replacement, increasing the latency of other misses to the same block (from other nodes) which have to wait to be processed until the L2 is unblocked.

However, it is not strictly necessary that the L2 attends all requests sequentially. In particular, it is easy to modify *List* so that read misses are attended immediately by the L2 even when it is blocked due to a replacement as long as the replacing node is not the first of the list. *Concurrent replacements* allow the L2 to process read requests concurrently to replacements, while other request combinations still need to be handled sequentially. Specifically, only one read request can be processed concurrently to one replacement at the same time.

This is possible because new sharers are always inserted at the beginning of the list in the *List* protocol. This means that, once the L2 has forwarded a replacement request, new nodes can be inserted in the list without risk because

it would be impossible for them to be the predecessor of the replacing node, which is what the forwarded replacement message seeks.

The replacement of the first sharer cannot be done concurrently to an insertion due to races, no matter whether the read request or the replacement request is received first. As a consequence, to be able to detect this case, the L2 needs to update its pointer to the first sharer only when the insertion transaction is completed (i.e., after receiving the *Unblock* message). Note that this limitation is not important in practice because it is not necessary to actually traverse the list to replace the first sharer.

This optimization requires the addition of a new intermediate state to the L2 coherence controller which combines the intermediate states that deal with replacements and insertions. It improves miss latency by reducing the waiting time of requests at the L2 controller.

4 Evaluation results

4.1 Simulation methodology

We have done the evaluation of the cache coherence protocols mentioned in this work using the PIN [12] and GEMS 2.1 [13] simulators, which have been connected in a similar way as proposed in [14]. PIN obtains every data access performed by the applications while GEMS models the memory hierarchy and calculates the memory access latency for each processor request. We model the interconnection network with the Garnet [1] simulator. The simulated architecture corresponds to a single chip multiprocessor (*tiled*-CMP) with 64 cores. The most relevant simulation parameters are shown in Table 1.

Table 1. System parameters.

Memory parameters	
Block size	64 bytes
L1 cache (data & instr.)	32 KiB, 4 ways
L1 access latency	1 cycle
L2 cache (shared)	256 KiB/tile, 16 ways
L2 access latency	6 cycle
Cache organization	Inclusive
Directory information	Included in L2
Memory access time	160 cycles
Network parameters	
Topology	2-D mesh (8×8)
Switching and Routing	Wormhole and X-Y
Message size	4 flits (data), 1 flit (control)
Link time	2 cycles
Bandwidth	1 flit per cycle

We evaluate *List* with the two proposed optimizations (*OR* and *CR*), and we compare it to two protocols using centralized sharing codes. The first one, namely *1-Pointer*, resembles the AMD's *MagnyCours* [4] protocol and uses a single pointer to the owner as sharing information (therefore having similar area requirements than *List*). The other one, *BitVector*, employs as sharing code non-scalable bit-vectors in each directory entry.

Our simulations consider representative applications from both the SPLASH-2 [18] and the PARSEC 2.1 [3] benchmark suites. *Barnes*, *Cholesky*, *FFT*, *Ocean*, *Radix*, *Raytrace*, *Volrend*, and *Water-NSQ* use 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 [2]. To do so, we have performed a number of simulations 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.

4.2 Results

L1 cache miss latency. L1 cache miss latency is a key performance aspect. The sharing code employed by the protocol can affect it significantly, specially for large core counts.

Figure 3 plots the average L1 miss latency split in five parts: the time spent in accessing the L1, accounting for stalls due to on-going coherence actions or exhausted MSHR capacity (*At_L1*); the time from L1 to L2 to access the directory information (*To_L2*); the time spent waiting at L2 until it can attend the request, mostly because of on-going transactions on the same block (*At_L2*); the time spent waiting to receive the data from main memory in case the block is not on-chip (*Main_memory*); and the time since the L2 sends the data or forwards the request until the requester receives the missing block (*To_L1*).

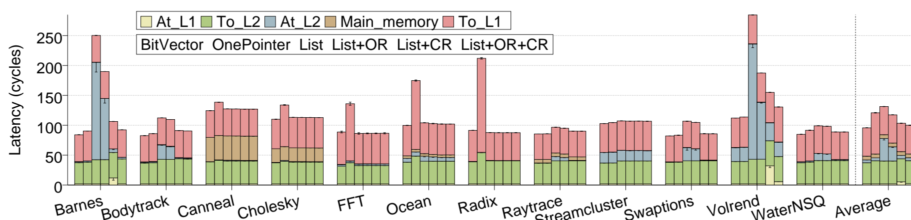

Fig. 3. L1 miss latency.

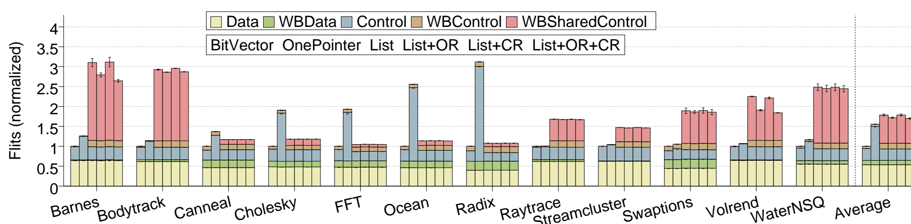

Fig. 4. Interconnection network traffic.

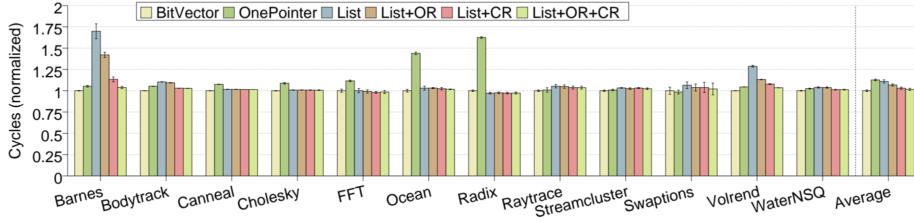

Fig. 5. Execution time.

List experiences an increase in latency compared to the area-demanding *BitVector*, mainly because of the sharp increase in the *At_L2* latency. In effect, *List* “blocks” a memory block when updating the sharing list to ensure mutual exclusion and to avoid inconsistencies in the list. This forces the delay of subsequent cache misses to the same block. On average, *1-Pointer* experiences a significant but smaller increase in latency due to the broadcast that it requires to invalidate sharers upon write misses.

Replacements of shared blocks in a private cache in *List* require to sequentially traverse the list of sharers. Differently, in *BitVector* and *1-Pointer*, these replacements are performed silently, without accessing nor blocking the L2 entry. Opportunistic replacements allow several replacements to happen at the same time, thus reducing the *At_L2* latency, notably in *Barnes* and *Volrend*. Concurrent replacements allow to resolve one read miss while performing a replacement, which further reduces miss latency down to a similar value as with *BitVector* and smaller than *1-Pointer*, making *List* a competitive protocol in terms of performance.

Despite the serial nature of invalidation in linked protocols, the *To_L1* latency is not affected. This counter-intuitive result is due to the low frequency of writes misses (23%, on average), but most importantly to the low number of sharers found upon such write misses (54% none, 41% one, 3% three, on average).

Network traffic. Figure 4 shows the network traffic, measured in flits, for each protocol. Traffic has been normalized with respect to *BitVector* and divided in the following categories: data messages due to L1 misses (*Data*); data messages due to L1 replacements (*WBData*); control messages due to L1 misses (*Control*); control messages due to L1 replacements of private data (*WBControl*); and control messages due to L1 replacements of shared data (*WBSharedControl*).

1-Pointer increases traffic with respect to *BitVector* in more than 50% due to the use of broadcast for invalidating sharers. *List* notoriously increases the traffic due to replacements of shared blocks in *Barnes*, *Bodytrack*, *Raytrace*, *Streamcluster*, *Swaptions*, *Volrend*, and *WaterNSQ* with respect to *BitVector* since in *BitVector* these replacements are silent and do not generate traffic. This increase in traffic is even slightly larger than the increase suffered by *1-Pointer*. Oportunistic replacements reduce the *WBSharedControl* traffic by coalescing replacements.

Execution Time. Given the previous results, small differences in execution time are expected when distributed shared codes are employed. In effect, as shown in Figure 5, *List* and *1-Pointer* both incur a similar slowdown of approximately 12% with respect to *BitVector*.

However, the optimizations described in Section 3 are able to almost completely eliminate this slowdown when both are combined. *List+OR+CR* incurs in an average execution time degradation of 1.6% with respect to *BitVector*, with a maximum reduction in execution time of 2.7% (*Radix*) and maximum degradation of 3.8% (*Barnes*).

Scalability. Three aspects reflect the scalability of a protocol: directory memory overhead, latency of cache misses, and network traffic. Since the first aspect has been addressed in Fig. 1, we now focus on the latter two. The latency of read misses in linked protocols is independent of the system size. However, the latency of write misses increases with the number of sharers per invalidation. Fortunately, this number grows more slowly than the system size (e.g., we have measured 0.52 sharers on average for 16 cores and 0.57 for 64 cores). Regarding traffic, the overhead comes from L1 replacements. In *List*, the number of messages per replacement depends on the size of the list which, as already mentioned, does not increase as fast as the total number of cores.

5 Conclusions and opportunities

In this work, we show that a singly-list based linked protocol (*List*) has the potential of providing simutaneously both scalable directory memory overhead and high performance. We find out that the most remarkable source of inefficiencies in *List* is the lack of silent replacements for clean data (although some other proposals do not make use of this important advantage to reduce network traffic and directory controller activity) and we have presented two techniques (OR and CR) that remove completely the increase in execution time that otherwise would emerge. Regarding the disadvantages typically associated to these protocols, we see that the impact that list traversal operations have on average cache miss latency is minimal. This is because read misses are more frequent than write misses and because the number of sharers that need to be invalidated on every write miss (the length of the list to be traversed) is usually small. Also, *List* only requires the addition of one pointer to each private cache (although modifications to the private caches could be avoided by having per-core *pointers caches*).

All in all, we show that a simple implementation of a linked protocol is an interesting alternative to current proposals in the maycore arena, and therefore, linked protocols can constitute an attractive starting point for proposing further optimizations for (for example) reducing cache miss latencies and thus going beyond the performance of *BitVector*. We are currently exploring this direction.

References

1. Agarwal, N., Krishna, T., Peh, L.S., Jha, N.K.: 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). pp. 33–42 (Apr 2009)
2. Alameldeen, A.R., Wood, D.A.: Variability in architectural simulations of multi-threaded workloads. In: 9th Int'l Symp. on High-Performance Computer Architecture (HPCA). pp. 7–18 (Feb 2003)
3. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: 17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). pp. 72–81 (Oct 2008)
4. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., Hughes, B.: Blade computing with the AMD Opteron™ processor ("Magny Cours"). In: 21st HotChips Symp. (Aug 2009)
5. Cuesta, B., Ros, A., Gómez, M.E., Robles, A., Duato, J.: Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In: 38th Int'l Symp. on Computer Architecture (ISCA). pp. 93–103 (Jun 2011)
6. Culler, D.E., Singh, J.P., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, Inc. (1999)
7. Demetriades, S., Cho, S.: Stash directory: A scalable directory for many-core coherence. In: 20th Int'l Symp. on High-Performance Computer Architecture (HPCA). pp. 177–188 (Feb 2014)
8. Fang, L., Liu, P., Hu, Q., Huang, M.C., Jiang, G.: Building expressive, area-efficient coherence directories. In: 22st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). pp. 299–308 (Sep 2013)
9. Fernández-Pascual, R., Ros, A., Acacio, M.E.: Characterization of a list-based directory cache coherence protocol for manycore cmps. In: 3rd Workshop on On-chip Memory Hierarchies and Interconnects (OMHI 2014). pp. 254–265 (Aug 2014)
10. James, D., Laundrie, A., Gjessing, S., Sohi, G.: Scalable coherent interface. Computer 23(6), 74–77 (1990)
11. Lovett, T., Clapp, R.: STiNG: A cc-NUMA computer system for the commercial marketplace. In: 23rd Int'l Symp. on Computer Architecture (ISCA). pp. 308–317 (Jun 1996)
12. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). pp. 190–200 (Jun 2005)
13. Martin, M.M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Computer Architecture News 33(4), 92–99 (Sep 2005)
14. Monchiero, M., Ahn, J.H., Falcón, A., Ortega, D., Faraboschi, P.: How to simulate 1000 cores. Computer Architecture News 37(2), 10–19 (Jul 2009)
15. Sanchez, D., Kozyrakis, C.: SCD: A scalable coherence directory with flexible sharer set encoding. In: 18th Int'l Symp. on High-Performance Computer Architecture (HPCA). pp. 129–140 (Feb 2012)
16. Thapar, M., Delagi, B.: Stanford distributed-directory protocol. Computer 23(6), 78–80 (1990)
17. Thekkath, R., Singh, A.P., Singh, J.P., John, S., Hennessy, J.L.: An evaluation of a commercial cc-NUMA architecture: The CONVEX Exemplar SPP1200. In: 11th Int'l Symp. on Parallel Processing (IPPS). pp. 8–17 (Apr 1997)
18. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: 22nd Int'l Symp. on Computer Architecture (ISCA). pp. 24–36 (Jun 1995)