J. Parallel Distrib. Comput. 102 (2017) 163–174

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

An adaptive cache coherence protocol: Trading storage for traffic

Lucia G. Menezo *, Valentin Puente, Jose-Angel Gregorio

University of Cantabria, 39005, Santander, Spain

HIGHLIGHTS

- A new adaptive non-inclusive cache coherence protocol.
- Combination of snoop-based and directory-based coherence protocol.
- Non-inclusive directory able to reconstruct sharing information when needed.
- Adaptive filter to minimize coherence traffic.

ARTICLE INFO

Article history:
Received 12 February 2016
Received in revised form
15 November 2016
Accepted 18 December 2016
Available online 28 December 2016

Keywords:
Coherence protocol
Multicore
CMPs

ABSTRACT

This paper introduces a new adaptive cache coherence protocol which minimizes energy requirements and guarantees scalability. It includes two complementary parts: a non-inclusive sparse-directory to track only actively shared blocks and a structure to determine the presence of a block in the private caches based on an improved counting bloom filter. It uses token counting to preserve the system correctness, to improve performance and to reduce the implementation complexity. Combining all these characteristics, the proposal has a low storage overhead and is able to suppress most of the traffic inherent to snoop-based protocols and reduce the size of directory-based structures. Using a capacity to track only 40% of all the blocks allocated in the private caches, this coherence protocol is able to achieve better performance than an over-provisioned sparse-directory with a capacity to track 160% of the blocks kept in private caches. The complementarity of both structures enables the coherence controller to change dynamically the way the storage available is dedicated according to the data-sharing properties of the application. Thus, applications with high-sharing degree will need more directory space while low-sharing degree patterns will need more private block-presence space to include more information. With only 5% of the private cache entries tracked, the average performance degradation is less than 8% compared to a 160% over-provisioned sparse-directory.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Cache coherence is a huge challenge of future chip multiprocessors (CMPs). In order to maintain the performance improvement, the number of cores will keep on increasing and the pressure on the bandwidth off-chip will continue to grow. One way of alleviating this problem comes by increasing the amount of memory on-chip. However, this increased amount of on-chip memory provokes longer access times, which need to be palliated using a more complex memory hierarchy.

It is the coherence protocol's responsibility to make sure that all potential copies of any block that are scattered over different caches are coherent, i.e. any processor should see the same content of all the memory locations under any circumstance. There is no universal solution and the chosen one will depend on the system. When the number of cores is low, the chosen solution is to use broadcast-based coherence protocols. Actually, this is the method used by current high-performance commercial systems [14,8,16]. Its main advantages come with better performance and lower complexity when compared with other coherence proposals such as directory-based coherence protocols. However, this is achieved with an increment in the total amount of traffic and cache snoops, which will decrease the energy efficiency of the system. It is clear that when the size of the system grows, the impact of this disadvantage will become unsustainable. A more subtle effect, but not less relevant, is the on-chip resource contention that characterizes these protocols. As a result, on-chip access latency can be affected, perhaps degrading the CMP performance under some particular usage scenarios.

* Corresponding author.
E-mail addresses: gregoriol@unican.es (L.G. Menezo), vpuente@unican.es (V. Puente), monaster@unican.es (J.-A. Gregorio).

http://dx.doi.org/10.1016/j.jpdc.2016.12.020
0743-7315/© 2016 Elsevier Inc. All rights reserved.

On the other hand, there are directory-based coherence protocols. Broadcast cache snoops are avoided by using a specific structure to track the block's copies present in the cache hierarchy. However, with this solution new limitations emerge. On the one hand, this approach demands inclusivity. This property requires that the contents of all the smaller caches of a multi-level cache hierarchy have to be a subset of the last-level cache (LLC). When a line is evicted from the LLC, inclusion is enforced by removing that line from all the caches in the hierarchy where it is present. Although from a performance and a cost stand point of view non-inclusiveness is desired, the common assumption is that inclusiveness is difficult to avoid in order to maintain coherence protocol complexity limited. On the other hand, the associativity needed in the directory increases as the number of cores does, making solutions like the duplicate-tag unviable [20]. The solution adopted to overcome this issue is to overprovision the directory to minimize unnecessary evictions in the private caches due to directory conflicts under constrained (and realistic) associativity [15]. However, this method also causes scalability problems as the private cache sizes increase and so the number of tracked blocks does too [35].

From this standpoint, it would appear that a pure coherence protocol might not be the most suitable approach to tackle the problem. Intuitively, it seems that the coherence protocol should somehow hybridize the best of both types: trying to attain the performance effectiveness and implementation cost of a broadcast-based coherence protocol with the energy efficiency of a directory-based one. This paper addresses this task and successfully attains a new coherence protocol, denoted FLASK (FiLtered and Allocated just by Shared block Keeper) coherence, which can scale as a directory-based coherence protocol does, while achieving cache effectiveness similar to a broadcast-based one. A previous version of this paper was presented in [28]. Now, several additional explanations with rewritten sections have been added to clarify the description proposal and a more realistic memory model has been used to produce new results, generating a complete and self-contained work.

FLASK is based on the idea of having two complementary structures working together on one logical framework. One of them, called *Dir-P*, which is basically an improved Bloom filter [6] to determine the presence or not of any block in the private caches. The other one, called *Dir-S*, works as a sparse directory and tracks the blocks that are actively shared among the cores. The whole system uses *token counting* to guarantee correctness while maintaining the complexity limited. The whole framework keeps detailed information about the location of data that are being actively shared while recording only the presence of the blocks that are allocated privately in the caches (with no sharers), which is the most common case.

In a directory-based protocol every block inside a private cache must be tracked by using an entry in the directory. This is known as *directory inclusivity* and it means that if a new block has to be tracked and there are no available entries in the directory, one of the existing ones has to be replaced. Thus, the block being tracked by the replace-to-be entry has to be invalidated in all the private caches where it is present. However, our protocol is able to handle incomplete information given by the framework, i.e. with the information of both structures, the protocol does not have exact information. This circumvents directory inclusivity and so it avoids the invalidation of blocks in the private caches due to conflicts in the directory structure. When a shared block is not being tracked by Dir-S, after a new request to the controller, a broadcast is sent to all coherence agents (the system structures in charge of the coherence maintenance). The replies are used to reconstruct the corresponding entry in the directory. This approach of reconstructing the directory entries on demand was first introduced by MOSAIC [27]. Nevertheless MOSAIC allocates directory entries for any on-chip miss (i.e. for both private and shared blocks) and it always generates a broadcast if there is a miss in the directory. On the contrary, the FLASK structure *Dir-P* includes the (probable) presence information which avoids the unnecessary search for the block inside the chip. If the block is not present inside the chip, i.e. *Dir-P* does not include the block's information, it sends the request directly to the memory controller. Thus, nearly all of the off-chip requests are not delayed. In the least common case, misses in a private cache of an actively shared block are always tracked by *Dir-P* and dealt with through a multicast to the on-chip coherence agents inside the chip, but avoiding unnecessary off-chip requests.

Finally, the framework introduced allows us to dynamically assign, according to the sharing degree of the running workload, storage capacity in the coherence controller either to track shared blocks in *Dir-S* or to identify privately held blocks in *Dir-P*. This characteristic enables the area dedicated to both structures to be reduced considerably.

The main contributions of the paper are as follows:

- The hybridization of a directory-based and a broadcast-based coherence protocol in a unified logic substrate with optimized implementation and energy costs.
- The proposed strategy achieves the performance of a conventional, over-provisioned sparse directory, while tracking less than 40% of the private cache entries. Similarly, it improves on Token coherence protocol performance by 10% and energy delay product by 20%.
- With only 5% of tracked private cache entries, average performance degradation is less than 10% with respect to a 160% over-provisioned sparse-directory.
- We show that, using an adaptive storage assignation at the coherence controller according to the workload properties, we can reduce even further the resources of the proposal. Matching a sparse-directory performance while tracking only 20% of private cache entries.

2. Motivation

2.1. Directory and broadcast coherence approaches

While directory-based protocols seem to be an attractive approach to enforce cache coherence in a CMP, when the number of cores is high and the on-chip hierarchy complexity grows, the directory is difficult to scale. The main cause is the large number of blocks that have to be tracked as the caches sizes grow. In an on-chip cache, similar to state-of-the-art systems [14,16,9], in order to close the gap in the access time between a small L1 (dominated by processor clock cycle) and a very large LLC (dominated by main memory access time), at least one intermediate level is required. As a consequence the number of blocks that the directory has to track is even larger. Additionally, those intermediate levels usually have a substantial associativity. Moreover, recent designs [14,16] also require a large associativity for L1. In summary, the number of blocks that can be mapped in a set of the directory is high.

Although in some early CMPs [20] the directory has enough capacity to keep information about all the blocks allocated in the private caches, when the number of cores or private cache complexity and size grows, this is not feasible due to the enormous associativity required by the directory. However, reducing this associativity increases the eviction of blocks in the private caches due to conflicts in the directory. A rule of thumb [15] suggests that over-provisioning the directory with twice the capacity required to track the private caches will diminish the number of invalidations [11]. Nevertheless, the larger number of tracked

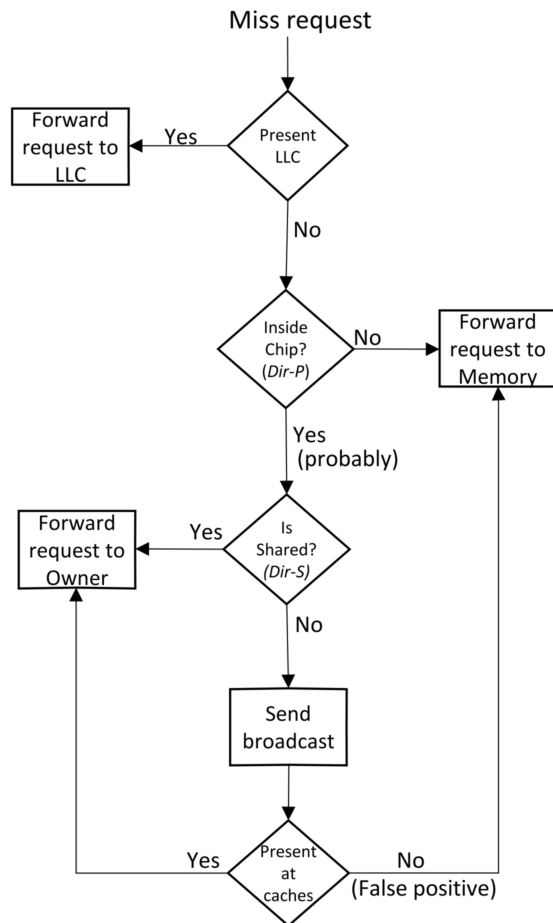

Fig. 1. Flow diagram followed by a miss request.

blocks as well as the size of the sharing vector cause a square growth of the directory cost with the increase in the number of cores in the CMP.

One way of handling this over cost in the directory is to consider the semantics of the applications. It is known that most memory regions are accessed privately by a single core most of the time [10,3]. If the directory is aware (actively [10,3] or passively [27,11]), we can reduce the number of private blocks that we have to track in the directory. Consequently, we can reduce the number of entries without interfering with the private cache performance.

Additionally, with a restricted number of cores, broadcast-based coherence approaches seem to be the most suitable choice. They do not need any type of structures to track the cache blocks' information and their performance is better than the directory-based ones under restricted circumstances. Proof of this is that many commercial high-performance processors use them [14,16,9]. However, similarly to the directory, when the number of cores increases, their scalability is negatively affected due to additional traffic and to the extra cache snoops that each cache miss triggers in the rest of the caches.

In this case, one way to tackle the problem is to use suitable interconnections that minimize the utilization of the same resource in the network by copies of the same message. This will be done by supporting on-network broadcast and/or on-network gather [18,21]. At the same time, to avoid both communication and tag snoop overheads, many works advocate filtering [9,29] or adapting the protocol behavior to the bandwidth availability [26,31]. In order to filter out unnecessary memory controller (MC) accesses or off-chip coherence fabric interfaces (XC), additional mechanisms should be provided [16,9]. In order to reduce the directory overhead, Bloom filtering mechanisms or similar ones are used. For example, to filter coherence messages to eliminate needless lookups [23]; to avoid having to store the tag in the directory [40,41]; or by using hash functions to reduce its size [35,13,34].

2.2. Cache coherence hybridization

Many of the previous solutions proposed to alleviate the limitations of both types of protocols, directory and broadcast, are based on a complementary design alternative. For example, in a directory for shared blocks, the approach followed is to snoop all or a subset of the coherence agents to see whether they have a copy of the block when a request misses in the directory and/or LLC [11,27]. In other works, the coherence protocol acts as a snoop-based protocol does. Therefore, the resulting protocol mimics a directory protocol in some cases and a broadcast-based protocol in others. The same observation could be made for some broadcast-based coherence protocols, which reduce the energy overhead through the insertion of structures to filter unnecessary cache snoops [29] or, for a different target architecture, predicting multicast destinations and using the directory to verify the selection [4]. In some way, most of these solutions use a "base" approach (either directory or broadcast) and use the complementary one to compensate for the inherent limitations. Similarly, our proposal combines both strategies: a standard sparse directory [15] and token coherence [24]. We will use a directory-like structure to track most of the shared blocks (with precision in a sharing vector) and private block presence (in an approximate way). In both cases, token coherence is used to discover when, after a miss, a block should be classified in one or other group. The information present in this structure will be used to minimize the on-chip and off-chip traffic. Then, intuitively, we can say that both facets of the coherence protocol operate with similar levels of relevance.

3. Flask coherence

3.1. Cache coherence protocol: basic idea

The coherence controller is based on a similar structure to an N-way directory, but divided in two different parts: *Dir-P* and *Dir-S*. The *Dir-P* or Private structure (*P* could also be considered for *presence* in the private caches) contains the basic information about whether a block is present in the private caches, either being shared or not. It is based on a *d-left Counting Bloom Filter* (dlCBF) [6]. The other part, *Dir-S* or Shared, contains detailed information about the sharers of all or *some* of the blocks that are being shared. Its structure corresponds with a sparse directory [15]. In both cases, the information does not have to be exact. In the *Dir-P*, the protocol admits false positives so the structure might say that the block is present in the private cache, when really it is not. In the *Dir-S*, false negatives can happen so the absence of any information about a block does not mean that is not being shared.

In summary, when a block is allocated in some private cache, its presence should be detected in the private part of the directory (*Dir-P*), while if the same block is being shared among several cores, then the shared part of the directory (*Dir-S*) could include its sharing information (but it is not mandatory). In the case that a block is only present in the LLC, then it is not necessary for the directory to include any information about it in either of the two parts (it is not a private block or a shared block).

The logical process (not the real one) that a core miss request goes through can be seen in the flow diagram in Fig. 1. If the block is present in the LLC, then the request should be forwarded there so it can be dealt with. If it is not, then it is necessary to check whether the block is inside the chip, i.e. if it is allocated in any private

cache. If it is not, then the request is sent directly to memory so it can forward the necessary data back to the requestor. If the *Dir-P* structure gives a positive answer then it is probable that the data is present in some private cache (but not sure). The next fact to check is whether the *Dir-S* includes information about the requested block, which would mean that the block is being actively shared among the cores. If it is, then the directory controller will know exactly who to send the request to in order to be able to deal with it. On the contrary, if the *Dir-S* does not include an entry with the block information, then the directory controller has to distinguish two different situations: (a) the *Dir-P* gave a false positive and the block is not really inside the chip, or (b) there is no entry because either *Dir-S* did not have enough space for that shared block or the block is privately allocated in some private cache. In any case, the directory controller sends a broadcast request to all the private caches asking for information about the requested block. If no private cache has the block allocated, then it is a case (a) situation and the controller detects the false positive given by *Dir-P*. As soon as the controller confirms that the block is actually inside the chip (case b), it forwards the request to the correct sharer and it allocates a new entry in the *Dir-S* for the address requested, setting all its sharers.

3.2. Update and replacement

When a block is sent from memory and enters the chip for the first time the *Dir-P* information for that address is updated to indicate that the block is present in the chip. The *Dir-S* information is still not modified, as the block corresponds to a private block which will be held by only one core (the one requesting it). From the moment that a private block is requested by another core, a new entry has to be allocated in the *Dir-S* since the block is then considered to be an actively shared one. If this new entry causes a conflict with another one already present, the old one is evicted silently, without notifying the sharers of the block as a conventional directory would do. This means that even if all the sharing information is lost, the sharers will not have to invalidate their copy of the block, because the coherence protocol is prepared to 'reconstruct' the sharing information in case it is needed in the future. Since directory inclusivity is not mandatory, unnecessary invalidations of the blocks present in private caches are avoided.

Last, when a block is replaced from all the private caches, the coherence controller removes the corresponding information in the directory. It removes the entry from the *Dir-S* if it is present, and removes the address information from the *Dir-P* since the block is not present in the private caches anymore. Any subsequent request for that address will find the data block in the LLC or in memory.

3.3. Flask *basic framework*

To achieve the coherence controller functionality that has been previously described, it is necessary to implement the two different functional parts of the directory-like structure: *Dir-P* and *Dir-S* Fig. 2. For the *Dir-P* part, and due to the large number of blocks that have to be tracked, Flask favors efficiency and sacrifices determinism by using a probabilistic structure. The chosen solution is a *d-left Counting Bloom Filter* (dlCBF) [6] that at least doubles the efficiency of the counters of a conventional Bloom Filter [5], with a similar implementation cost. For the other part, the *Dir-S*, where most of the blocks that are being shared have to be stored, the chosen structure is an *n*-way sparse directory [15].

In order to obtain a coherence protocol which is adaptive to the application necessities, both structures need to be independent of the storage space available and the performance of the protocol should not depend on it. In fact, it is possible to reduce to zero the storage space available and the coherence protocol would still work. Reducing each of the structures in the directory separately does not have very different consequences. As the area dedicated to the *Dir-P* is reduced, the number of false positives will increase and therefore will also increase the number of broadcasts needed to check whether the block is present in any private cache. When this *Dir-P* area is null, any check into this structure will give a false positive and only the presence of the block in LLC will avoid the exhaustive search for the block. A similar thing happens as the area dedicated to the *Dir-S* is reduced. The number of shared blocks whose sharing information is not available at the directory becomes larger and consequently more broadcasts will be necessary to obtain the data block. If the system did not have any storage space at all, every request that does not find the data in the LLC would cause a broadcast. On the contrary, if both structures are well set, only the blocks that change from private to shared will require a broadcast to all the cores in the CMP.

This independency gives the Flask coherence protocol the possibility of completely adapting the area available to the necessities of each of the applications according to their sharing degree.

3.4. Broadcasts. Token coherence

With all the previous framework characteristics, it seems obvious that the coherence protocol is able to trade storage space for traffic. This introduces new possibilities in the CMP as the area used to store presence or sharing information can be adjusted to the available resources of the interconnection network. Thus, if the area to keep the blocks' information is large, the necessity to broadcast requests is considerably reduced. On the other hand, as the amount of block information decreases, the requirements of the interconnection network resources will increase.

In any case, the broadcast mechanism should be optimized to minimize its negative effects. This can be done in two ways. From the physical point of view, the existence in the interconnection network of special support for broadcast traffic can reduce the load on the network by up to 70% [18] of what is generally accepted, making the broadcast process less costly in time and energy. From a coherence point of view, the broadcast technique introduced by the Token Coherence protocol [24] presents many advantages over other proposals. This technique is based on the assignment of a fixed number of tokens to all the cache blocks (usually the same number as the number of cores in the system). To read a block, it is necessary to have a copy of the block with at least one of the tokens. To write a block, it is necessary to have a copy of the block and the set of all the tokens assigned to that block. Thus, the single writer/multiple reader invariant is guaranteed. This mechanism considerably reduces the complexity of broadcast-based coherence protocols and, specifically for our protocol proposal, offers three notable positive effects: (a) it reduces the latency response of the write requests. When the coherence controller broadcasts a write request, the requestor only needs the response of those having tokens and not from all of the cores; (b) it helps *Dir-P* to determine when the last copy of the block is entering or leaving the private caches, making it easier to control; (c) as happened in Mosaic [27], the use of tokens makes it possible for the LLC to work as a filter of a large number of unnecessary broadcasts, because if any requested block is present in the LLC and it includes all the tokens, the coherence controller is sure that there is no other copy of the block in the whole chip. In this case, the request can be sent only to the LLC so it can forward the requested data block to the requesting core.

4. Implementation details

This section introduces the main aspects of both structures mentioned previously: *Dir-S* and *Dir-P*.

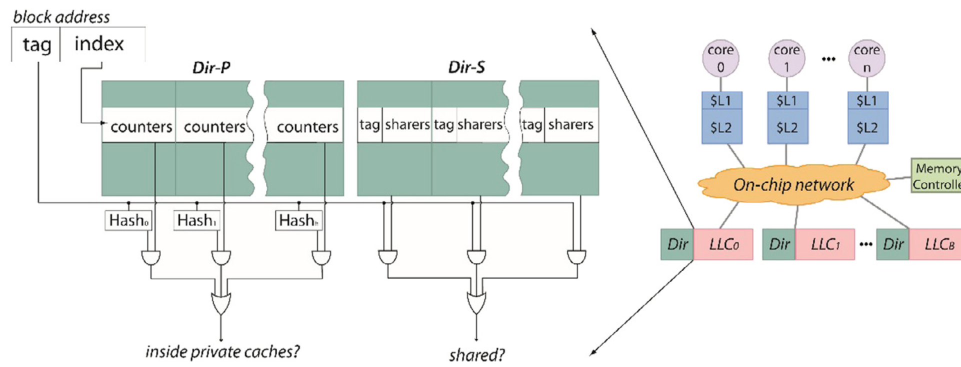

Fig. 2. High-level representation of a lookup in FLASK coherence controller (*Dir-P*: a parallel counting bloom filter with r counters and h hash functions. *Dir-S*: an N-way sparse directory).

4.1. Dir-S: sparse directory

In order to avoid using a broadcast to deal with all the private cache misses, FLASK uses a sparse-like directory *Dir-S* to track some of the shared blocks. In contrast to conventional sparse directories, this one will tend to track only actively shared blocks. This means that when a block is missing in the chip, no entry is necessary in this part of the directory. If the block remains private as it is replaced, it will be progressively moved to further levels until it gets evicted to the non-inclusive LLC. We denote the time between these two points the private caching period. The block becomes actively shared if during the private caching period it is accessed by another core in the system. If this circumstance arises, we need to allocate an entry in the *Dir-S* with its sharing information, i.e. with the two cores if the second request was a read, or with the last core if it was a write.

As was mentioned previously, when a *Dir-S* entry is evicted, no external invalidations are sent to the private caches that hold a copy of the block. Thus, when a processor misses in its private cache, it is not possible to determine whether there are other copies in other private caches just by checking if there is an entry allocated in the *Dir-S* or not.

At first, when a request arrives at the directory, and assuming for now that *Dir-P* says that the block is within a private cache and there is a miss in the *Dir-S*, the coherence protocol initiates a reconstruction process. This process will snoop the remaining coherence agents in order to recreate the sharing vector of the requested block. By using token counting, the directory will be aware of when the information received is enough to unblock the pending memory operation. The on-chip coherence agents will respond with the count of tokens owned for the requested data. The reconstruction approach is based on the one proposed in MOSAIC [27]. Nevertheless, FLASK never allocates an entry in the directory unnecessarily (i.e. for a private block).

If any response comes from any of the cores, i.e. from their private caches, we know that the block is being actively shared and so a *Dir-S* entry is allocated. If the pending operation is a load, the directory instructs the private cache with the owner token to forward the data to the requestor and it keeps counting incoming answers. When the location of all the tokens is known, the sharer vector is accurate. If the pending operation is a store, the directory instructs all the private caches with tokens to forward them to the requestor core (consequently invalidating the data) and also requests the cache where the owner token is allocated, to forward the data block to the requestor processor. When all the tokens are received by the requesting core, it knows that it can proceed with the operation, and unblock the directory entry. On the contrary, if the requesting data is not in any private cache, then the request is forwarded to memory and its response will not cause any allocation in the *Dir-S* as the data block enters the chip as a private block (its presence information will be included in the *Dir-P* part as is explained in the next section).

A miss in the *Dir-S* structure could happen for four reasons:

1. The entry allocated with the requested information was removed from the directory because of lack of space (remember that this is done silently).
2. The requested block is privately allocated in some private cache.
3. The block has been evicted from the private caches and it is present in the LLC.
4. The block is present in memory, outside the chip.

If no additional structure is added, any of these cases would cause a broadcast request to all the coherence agents in the system. If the block is present in main memory all the private cache tag snoops are an unnecessary overhead of traffic and energy. If the block is present in any of the private caches, it is not necessary to waste such a scarce resource as the off-chip bandwidth is with a memory request. Then, if we include a chip presence information structure like the one described next, the unnecessary private snoops and the off-chip requests could be reduced drastically.

4.2. Dir-P: D-left Counting Bloom Filter (dlCBF)

Dir-P is the structure in charge of determining if a block is present in any private cache or not. It is composed of counting bloom filters [5] attached to each *Dir-S* entry, in order to track all the tags (roughly) that map onto its entries. A counting bloom filter is an efficient, approximate set membership check that enables the tracking of a tag's presence in the private caches at a fraction of the regular cost. The storage space dedicated to this structure will determine its approximation degree. In any case, the coherence protocol must tolerate false positives which will become more frequent as the dedicated area is reduced.

Each time a block arrives at any private cache or leaves it, we need to increment or decrement the counters of the filter. The event that triggers the increment in the counters will be an on-chip miss. This situation can be identified when the data arrives at the chip after a miss. On the contrary, decrementing each counter might be a significantly harder task. It requires identifying when there is no longer a copy of the block in the private caches. We will again use token counting for this purpose. When a block has been evicted from every private cache, the directory knows that the block, with all its tokens, is in the LLC and the counters of the *Dir-P* can be decremented. This is the common case (>99.99%) because the reuse distance supported by private caches is much shorter than that supported by the LLC, given the size ratios between the two caches. Finally, the scenario in which a block that has to be evicted from LLC without having all the tokens could be very difficult to handle, but we opt for invalidating any private copy of the block through a broadcast, collecting the tokens, and evicting the block. This seems costly but, fortunately, it is very unlikely.

4.2.1. Counter overflow and false positives

As in any counting bloom filter structure, a problem we must deal with is that counters can overflow [7]. This situation might affect the system correctness since it could cause false negatives, i.e. requests sent to memory for blocks potentially modified in the chip. The coherence protocol should contemplate this event and handle it accordingly. As this is a very unusual case, it is not cost effective to increase the complexity of the coherence protocol because of it. Instead, we opt for avoiding counter saturation through invalidation. If after an on-chip miss (and subsequent addition), some of the counters reach the maximum value, we invalidate all the private cache sets mapped on *Dir-P*. Although at first sight this might seem to have a huge performance impact, in practice, with a large variety of configurations and workloads, we have seen that it is extremely unlikely that this event would ever happen. This is consistent with the fact that the probability of having a counter overflow, is very unlikely [12].

A second possible event that the protocol needs to be aware of is the false positives given by the *Dir-P*. Although they are quite unlikely if the structure is well dimensioned, they are not impossible. When there is a hit in the filter (*Dir-P*), most of the times it will mean that the block is inside the chip, becoming unnecessary to interrogate the memory. However, with a low but non-zero probability, it could happen that this hit is a false positive and the data block is not really inside the chip. If we proceed as usual, i.e. broadcasting request and waiting for an answer from token holders, the system might reach starvation because no one will reply. Instead of treating this situation through starvation detection [24], we prevent it from happening by forcing cores without tokens to acknowledge directory entry reconstructions. In contrast to other protocols [9], the overhead of these acknowledgments is only present when there is a hit in *Dir-P* (and a miss in *Dir-S*). The directory coherence controller will wait for all these negative acknowledgments before sending a request to the memory. For this reason, on-chip misses in which the counting bloom filter returns a false positive will be delayed until the coherence controller is aware of this block not being inside the chip. Consequently, memory accesses will be delayed by a few tens of cycles in the worst case. The positive effect of this approach is that the memory controllers do not have to be filter or cancel unnecessary memory requests [9], considering that the cost of this functionality might be substantial.

5. Storage efficiency

5.1. dlCBF representation

The storage capacity budget of the coherence controller should be used to keep block sharers in the sparse directory, *Dir-S*, and an indication of the presence of a tag in any private cache in *Dir-P*. For a conventional Counting Bloom Filter (CBF) with n private blocks mapped onto one entry and m counters with the optimal number of hashes, according to [7], the probability of a false positive is approximately:

$$P_{fp_CBF} \approx (\ln 2)^{m/n}.$$

For example, to achieve a false positive probability of 5%, we require that $m/n > 8$, which is equivalent to having at least 8 counters per private cache entry. To minimize the saturation probability, we need at least 4 bits per counter. In this case, the overflow probability is approximately $e^{-\ln 2}(\ln 2)^{16}/16! = 6.8 \cdot 10^{-17}$ [7]. Therefore, we need at least 32 bits per private block. This represents a saving of 50% for a 64-bit tag.

Unfortunately, conventional hash functions behave far from ideally, which unbalances counter usage in a CBF [32]. However, a quasi-perfect alternative is *d-left hashing* [39]. Over this base,

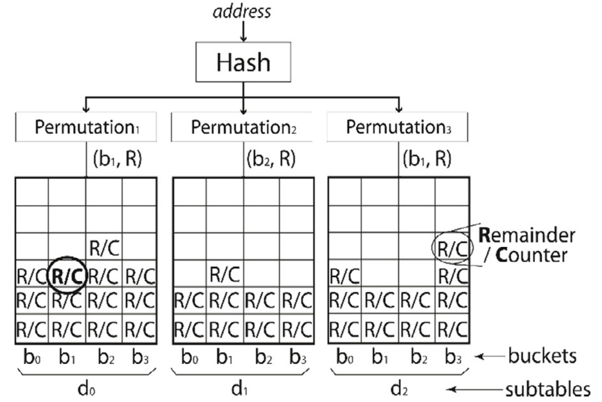

Fig. 3. Sketch of a dlCB filter.

Bonomi et al. [6] introduce a new structure called a *d-left* Counting Bloom Filter (dlCBF) that at least doubles the efficiency of the counters of a CBF, with a similar implementation cost.

Fig. 3 shows a high-level representation of this type of filter. It works as follows: the table is divided in d *sub-tables* ($d = 3$ in Fig. 3, but four in practice might be enough). Each table is divided into b *buckets* ($b = 4$ in our example) and each bucket is divided into multiple cells (six in our figure). These cells include a few bits to store the address signature R, called *remainder* and a small *counter* c. The counter is only used for the unusual case in which different addresses have the same remainder (in practice 2 or 3 bits suffice). For a corresponding address, to compute the bucket b_i and the remainder R, we apply a conventional hash function $H(\cdot)$ to that address, obtaining $\log_2 b + r$ bits and next we apply d permutations this value $\Pi_i(H(x))$. Each of these permutations will result in a pair (b_i, r_i) corresponding to the i sub-table. After computing the bucket for each sub-table, an entry is allocated in the least used bucket, i.e. the bucket with fewest elements. Here is where the key element of these structures, the d-left hashing, makes the difference. It states that for an address, when there are several destination buckets with the same number of entries, the left one should always be chosen [39]. This allows near-perfect cell load usage, and consequently better coverage for the filter.

Next, we will describe the process with a simplified example. We will assume that the d permutations are obtained with the function:

$$\Pi_i = aH(x) \mod 2^{(\log b + r)} \quad \text{with } odd\ a < 2^{\log b + r}.$$

If we assume only 2 buckets for each sub-table (which means 1 bit for the counter) and 3 bits for the remainder, the result of $p = H(x)$ must be a number between 0 and 15, where the most significant bit indicates the bucket and the rest will be the remainder. Considering $p = H(x) = 6$ and choosing three odd values (one for each of the sub-tables: 3, 7 and 11), the resulting values of applying the permutations are as follows:

$$P_0 = 3 \cdot 6 \cdot \mod 2^4 = 2 -> bucket\ 0, remainder\ 010$$
$$P_1 = 7 \cdot 6 \cdot \mod 2^4 = 10 -> bucket\ 1, remainder\ 010$$
$$P_2 = 11 \cdot 6 \cdot \mod 2^4 = 2 -> bucket\ 0, remainder\ 010.$$

Then, bucket 0 would be chosen for the first and third sub-table and bucket 1 for the second one. The decision would depend on the occupation and in the case of a tie, it would always be the one on the left (bucket 0 and sub-table 0). Note that for any addresses $y \neq x$ so that $H(y) = q$, if $q \neq p$, the permutation application will give as a result a different pair (bucket, remainder). In the example, $P_0(q) \neq 2, P_1(q) \neq 10yP_2(q) \neq 2$. The exception happens when both address have the same fingerprint, in which case the counter is increased.

One important aspect is that if invertible permutations are chosen, any value from any of the d sub-tables may be moved to any of the $d-1$ remaining. It is a similar process to what it is done in the Cuckoo directory [13]. For instance, if the datum of the previous example was put in bucket 0 of sub-table 2, it could be moved to bucket 0 of sub-table 0 or to bucket 1 of sub-table 1. The practical consequence of this is that the whole structure can be adaptive. Thus, a sub-table may be added with an additional permutation or any sub-table could be discarded by moving each of its elements to the "other" possible choices.

Looking up in the structure is a similar operation. Note that the total number of bits per bucket is pretty small (about 16), so the search can be done in an external register avoiding the need for a content addressable memory. Since all sub-tables are handled independently, like in a parallel bloom filter, we can use a conventional single-ported SRAM memory. Therefore, like in a conventional CBF, we can use the same storage to keep either *Dir-S* or *Dir-P* information. The controller logic should route each access in accordance with its use. Note that *Dir-P* updates are done on LLC replacement or on-chip misses; in the former case, outside the critical path, and in the latter one, overlapped with main memory access. Since the directory is banked, even in the case of having thousands of pending on-chip memory operations, the number of pending operations per bank will be low. Consequently, we can assume that the cost of updating *Dir-P* will be negligible. The look-up of this structure with d sub-tables is similar to a conventional parallel counting bloom filter with d hash functions: each bucket in the corresponding sub-table is read in parallel. We should look up the presence of the remainder in the bucket. This is equivalent to a tag search in a d-associative cache so we will assume that the look-up operation is the same as the one required for an equivalent directory.

5.1.1. False positives and counter overflow probabilities

For a dlCBF structure, the probability of false positives, with a d sub-table configuration of b buckets each, r bits per remainder and n private blocks mapped onto one entry, is given by [6]:

$$P_{fp_dlCBF} = 1 - \left(1 - \frac{1}{d \cdot b \cdot 2^{-r}}\right)^{n} \approx \frac{n}{d \cdot b} 2^{-r}.$$

Assuming cl cells per bucket and a utilization factor of $\rho(< 1)$, and choosing a total number of buckets depending on n as $n/(\rho \cdot cl \cdot d)$:

$$P_{fp_dlCBF} \approx \rho \cdot cl \cdot d \cdot 2^{-r}.$$

Thus, assuming four sub-tables ($d = 4$), eight cells per bucket ($cl = 8$) and a 3/4 usage per bucket ($\rho = 0.75$), to achieve a 5% false positive probability, the number of bits per remainder is 9. Considering a 3-bit counter, each cell requires 12 bits. To compensate for bucket utilization, we have to multiply this number by $1/\rho$, giving approximately 16 bits per element tracked. This is less than half the number required by the conventional bloom filter mentioned above.

In practice, for a 64-bit physical tag (which might include sharing vector and block state), this will lead to a 77% saving. For a 256 kB L2 and 32 kB L1I, 32 kB L1D and block sizes of 64 bytes, we will require ~9.2 kB per core (less than 3% of the tracked cache size), whereas a conventional CBF will require about 20 kB. For this configuration each bucket uses 128 bits, which simplifies the table lookup. If we reduce the filter size (to save area), the false positive probability might increase, but not in a significant way. Note that all the theoretical estimations are made assuming no spatial locality or sharing in the stream of addresses.

The counters associated with each remainder may suffer overflow. Although, as was mentioned before, the protocol includes a fallback mechanism to avoid any possible starvation situation, it is important to remark that the probability of this occurring is very low. For example, using a 3-bit counter with 5k of private blocks, a higher bound is given by [6]:

$$\binom{n}{2^c + 1}\left(\frac{1}{\#buckets \cdot 2^r}\right)^{2^c+1} \approx \binom{5000}{9}\left(\frac{1}{256 \cdot 2^9}\right)^{9} \approx 4.6 \cdot 10^{-19}.$$

5.2. Dir-P/Dir-S resource partitioning. Scalability

Each of the active entries allocated in *Dir-S* means a lower pressure on the *Dir-P*, because several copies of the same block in the private caches will only count as one element in the *Dir-P*. Taking this into account, we can state that the two structures are complementary. Restricting the *Dir-P* capacity will increase the probability of false positives, which will increase the on-chip traffic and the delay of the off-chip misses. On the other hand, restricting the *Dir-S* capacity to the point that the working set of the actively shared blocks does not fit in will increase the reconstruction probability. This could be used effectively to provide a low tracking capability (perhaps for less than 20% of private cache blocks).

As shown in Section 5.1, if the permutation used to fill the filter tables is invertible [6], then it is possible to reconstruct a sub-table by observing the remainder of any stored value and simply rolling back the permutation. Thus, it is possible to "move" the tracking of a block from one sub-table to another. In this way, if we detect that the *Dir-S* is highly loaded (through the frequency of reconstructions), we can adaptively de-allocate one of the sub-tables in *Dir-P* and expand the number of ways in the directory. Similarly, if the workload is not using the entries of the shared part because all/most of the data are private, we can expand the private part with additional sub-tables, reducing the false positives without increasing the reconstruction frequency. Note that this operation is not possible in a conventional CBF.

Initially, for all of our workloads, we will split the storage capacity equally between the two parts of the directory. Later, and after observing the application's behavior, we will explore workload-dependent partitioning in order to see how beneficial this approach can be for reducing adverse effects in extreme cases. We will not discuss implementation cost, although it does not seem to be an issue. Perhaps some multiplexors will be necessary to connect the SRAM content to the coherence controller. In any case, note that the dynamic behavior will have a negligible impact on performance since the one-by-one migration of *Dir-P* sub-table entries to *Dir-S* could be done in the background with the normal system operation. The opposite operation is direct (just requires invalidating directory entries that will be used as a sub-table). We will not discuss the mechanism for triggering this process since the sharing pattern of the applications is quite stable throughout the execution. The most cost-effective way is to trigger it by software at the beginning of the workload.

6. Evaluation methodology

6.1. System configuration

To analyze the proposal, we model a CMP with out-of-order cores that mimic the execution resources and on-chip cache hierarchy of the Intel Haswell processor [16]: using 6-wide issue cores with 196 in-flight instructions and up to 64 pending memory operations. The number of cores in the CMP is 16. Therefore, the coherence fabric has to support up to 1024 concurrent memory operations. There are three levels of cache. The first two are private, strictly non-inclusive (i.e. L2 acts as a victim cache of L1). The third level is shared and uses a mesh network, which is characterized by better on-chip bandwidth scalability than a ring network. We

Table 1
Summary of 16-core CMP system configuration.

Core Arch.	Functional Units	4xI-ALU/4xFP-ALU/ 4xD-MEM
	ROB size/Issue width	196, 6-way
	Frequency, count	3 GHz, 16 cores
Private caches	(L1) Size/Associativity/Block size/Access time/Repl.	32 kB I/D, 4-way, 64 B, 1 cycle, LRU
	(L2) Size/Associativity/ Block size/Access time/Repl.	256 kB Unified, 8-way, 64 B, 2 cycles, LRU, Exclusive with L1
	Outstanding requests per core	64
Shared L3	Size/Associativity/Block size/Repl.	16 MB (16 × 1 MB), 16-way, 64 B, LRU
	NUCA mapping	Static, interleaved by LSB
	Slice access time	6 cycles
Mem	Capacity/Memory controllers/ranks/banks	4 GB, DDR3-1600, 4 ranks, 16 banks (4 per controller)
Network	Topology/Link latency/Link width/Clock	4 × 4 Mesh, 1 cycle, 16 B, 3 GHz
	Router latency (low-load)/Flow control/Routing/ Buffering	1–3 cycle, Wormhole,DOR,10 kB

Table 2
Multithreaded workloads.

SERVER [2]	OLTP	IBM DB2 DBMS, TPC-C like 10 000 transactions
	Apache	Apache web server, SpecWeb like, 25 000 transactions
	JBB	SpecJBB, 70 000 transactions
	Zeus	Zeus web server, SpecWeb like, 25 000 transactions
NPB [19]	Multi-Grid (MG)	CLASS A
	Fast Fourier Transform (FT)	CLASS W
	LU Diagonalization (LU)	CLASS A
SPEC [37]	Astar	Native, 15 thr.
	Hmmer	Native, 15 thr.
	Omnetpp	Native, 15 thr.

will assume that the routers in the network can handle multicast traffic natively [18], have single-cycle low-load pass-through [22], separate virtual-networks to avoid end-to-end protocol deadlock and over-provisioned buffering (90 flits per port). Similarly to the LLC, the directory is banked and interleaved by the least significant bits.

In order to compare our proposal to snoop and directory-based protocols, we have implemented two reference protocols based on TokenB [24] and on a sparse-directory [15], respectively. TokenB has been selected because it allows a scalable unordered network to be used without adding extra mechanisms outside the coherence controllers. Using the same methodology and tools, all coherence protocols have been optimized fairly. A full SLICC (Specification Language for Implementing Cache Coherence) specification for a 3-level hierarchy can be found in [36].

The memory access has been modeled in a realistic way using four DDR3 controllers with four memory banks each. This is an important aspect for the confidence in the latency results of the simulation. In fact, we will see that in a realistic scenario, broadcast protocols should be handled carefully to avoid unnecessary memory requests wasting the scarce off-chip bandwidth resource [9]. A summary of the main parameters used in our analysis is shown in Table 1.

6.2. Workloads & simulation stack

We will use GEMS [25] as the main tool for our evaluation. With GEMS, it is possible to perform full-system simulations. Coherence protocols have been implemented using the SLICC language. In order to model accurately interconnection network contention and its impact on the average access time, we replace the original network with TOPAZ [1]. For power and cost modeling, we use CACTI 6.5 [30] for the cache and DSENT [38] for the network. Ten workloads, shown in Table 2, are considered in this study, including both multi-programmed and multi-threaded applications (scientific and server) running on top of the Solaris 10 OS. The numerical applications are three of the NAS Parallel Benchmarks suite (OpenMP implementation version 3.4 [19]). The server benchmarks correspond to the whole Wisconsin Commercial Workload suite [2]. The remaining class corresponds to multi-programmed workloads using part of the SPEC CPU2006 suite [37] running in rate mode (where one core is reserved to run OS services). The mix of workloads has been selected trying to cover diverse usage scenarios, varying the sharing degree (from none in SPEC applications to a large amount in Server Workloads) and sharing contention (from none in SPEC to a large amount in scientific applications). Among the NAS applications, we chose the three with the highest sharing contention. From the SPEC suite, we chose three applications with a variable range in working set size. We should emphasize that the three families of applications exhibit quite dissimilar behavior from the coherence protocol perspective, but they have to be considered, given the usage scenarios of general purpose CMPs. Focusing the evaluation on a single suite of benchmarks, it is hard to consider all the characteristics we have represented with the selected mix.

We model hardware-assisted TLB fill and register window exceptions for all target machines. Multiple runs are used to fulfill strict 95% confidence intervals (error bars are not visible in most cases). Benchmarks are fast-forwarded to the point of interest, during which page tables, TLBs, predictors, and caches are warmed up. In iteration-based applications, such as NPB, a warm checkpoint is taken in the middle of the execution and with a reduced number of iteration runs. Transactional workloads are warmed up by running hundreds of thousands of transactions, and accurately simulated for a fixed number of transactions. SPEC workloads are fast-forwarded to the point of interest and simulate ∼8 billion instructions.

7. Performance results

7.1. Comparative results with reference protocols

Figs. 4–6 show the fundamental parameters of the systems that have to be considered when running coherence protocols, namely execution time, average access-time and memory hierarchy and network energy delay product (EDP). With the aim of normalizing the total storage, we denoted it as SDE (Sparse-Dir Equivalent) capacity, i.e. the capacity to track all the entries of the private cache blocks (5 K SDE entries in the particular case of Table 1' values). Directory sizes to establish the comparative range from 8 K SDE entries (over-provisioning to track 160% of the private cache blocks) to just 256 SDE entries (about 5%). Obviously, to keep implementation cost constant, in FLASK this capacity is devoted to both *Dir-S* and *Dir-P* structures. At this point, we assume half and half, but different assignments will be discussed later.

As the reader might realize, in contrast to [28], there are no results corresponding to token coherence protocol for some benchmarks. This occurs because we are modeling a realistic memory controller where each of the requests sent to memory is actually using memory bandwidth. In spite of using a well-dimensioned network (see Table 1) and routers with state-of-the-art features [18,22], since the token coherence protocol sends a request to memory for each of the misses in the private caches, the memory controller saturates. The queues grow to a level when the memory is unable to send the data to the requesting node before reaching the deadlock threshold of the system.

Even modifying the number of retries before issuing *persistent requests* (the mechanism of token coherence protocol for starvation avoidance) or increasing the fixed timeout before issuing a retry broadcast, the latency of some packets reaches unrealistic values. This unstable behavior caused by the mechanism for starvation avoidance (persistent request) under realistic conditions has been reported by other authors [33]. However, this protocol is important for comparison because under unlimited bandwidth it provides the best performance.

As expected, when the directory size is below one fourth of the aggregate private caches' capacity, the sparse-directory performance is degraded. This degradation is shown after a significant directory size reduction. Even though in other workloads this degradation is seen sooner [11], in this case we have two levels of private caches where the second one acts as a victim cache for L1. Thus, a directory-induced private miss is eight times more frequent in L2 than in L1. Considering that the block reuse in L2 is low [17], the results seem reasonable. We could obtain more noticeable effects with an inclusive L1/L2, but still not too acute. Besides, having such a configuration for an aggressive out-of-order core and a shared LLC does not seem very interesting. Additionally, it should be noted that for an in-order core, any directory-induced private miss will have more effect on performance. The memory level of parallelism present in the evaluated system allows the impact to be partially hidden.

As Fig. 5 shows, the number of hits in private caches is reduced when the count of tracked blocks is decreased only in the sparse directory protocol. Thus, there is a substantial latency increment with this protocol as data must be retrieved from LLC (for private blocks) like in numerical or multi-programmed workloads, or from other private caches (for actively shared blocks) like in server workloads. These results are consistent with the applications' sharing degree. For applications with a large portion of shared data, the latency degradation when reducing the directory size from 160% to 5% is even doubled, degrading their performance by more than 40% on average. For these cases the intense coherence traffic due to directory-induced invalidations and subsequent LLC hits makes the activity in the network increase substantially, degrading the energy properties.

Figs 4, 5 and 6 enable the comparison of our proposal behavior to the other two protocols mentioned. As the storage capacity is changed from having an overprovisioned directory of 160% of a SDE to only 5%, Flask shows a much better behavior than its counterparts. On the one hand, the average performance degradation is only 10% compared to 40% seen in the sparse directory. On the other hand, although part of the Flask implementation is based on the token coherence protocol, it does not show any of its anomalies. Since there are no starvation issues, Flask does not need persistent requests. Besides, even in the most extreme configuration (5%) there is no saturation when using realistic memory controllers, and the applications with low sharing-degree barely show any performance degradation. Obviously, when *Dir-S* only has capacity to track 256 private blocks per core, there are more reconstructions which delay the access to shared data after a miss in the private caches, slightly lengthening LLC access time. In the multi-programmed workloads, in spite of having a minimal *Dir-P* structure (just 32 buckets per sub-table), the effects of this extreme configuration both on performance and energy are negligible.

As shown in Fig. 6, the energy requirements of Flask protocol are smaller to Token (in those applications where this does not collapse) and are quite steady regardless of the directory size. Note that this metric is pessimistic, since we exclude the cores' power consumption. As Flask is the best performer in most cases, the EDP of these components will be low.

In summary, with almost no tracking capability, Flask is more stable and energy efficient than broadcast-based protocol, and its performance degrades much more gracefully than conventional sparse-directory when directory size is reduced.

7.2. Reducing broadcasts. false positives

The performance advantage of Flask over the other options mainly comes from the reduction of broadcasts compared with snoop-based protocols while maintaining their efficiency. A part of this reduction corresponds to the *Dir-S*. Access to actively shared blocks is directly obtained looking up in this structure. Obviously, the smaller size of *Dir-S* compared to the sparse directory is paid by broadcasts to find the data. However, a significant number of them are unnecessary (if the requested data is not in the private caches) and therefore should be eliminated. This is precisely the task of the *Dir-P* structure. However, again because of the limited size of the structure, not all unnecessary broadcasts are eliminated. These are the *false positives* that unnecessarily increase on-chip energy and memory access time because they delay the memory request until the coherence controller realizes that there is no on-chip copy present.

The use of a real memory controller invalidates part of the results of [28] concerning the TokenB coherence protocol as shown in Figs 4, 5 and 6. Nevertheless, this protocol has to be considered as an important counterpart because, when considering no bandwidth limitations, it is the one that offers the best performance. Since the amount of traffic is not affected by the memory latency and attempting to provide a comparison among the three protocols for all the benchmarks, we will continue this analysis with an *ideal* memory controller. Thus, we can compare Flask's results with the number of multicasts done in TokenB, which are an upper limit.

Under these conditions, Fig. 7 shows the total number of multicasts sent by Flask with the number of broadcasts that were avoided by *Dir-P*, all normalized to the total number of broadcasts sent by TokenB. It must be taken into account that, if the network is able to handle multicast traffic and the cache snoop energy contribution, this might not be directly transferred to link utilization or energy consumption, as we can contrast these results with Fig. 6.

When the directory is dimensioned for 20% of the private cache capacity, the number of false positives seems to be consistent with the theoretically expected proportion of 5%. When we shrink the SDE capacity to just 5% of the private caches, different behaviors can be observed. In some cases, even with such a small size the number of on-chip misses detected is quite reasonable. Concerning the number of true positives, i.e. private cache misses for actively shared data, which are compulsory broadcasts; these vary among the applications. In some of them they grow when we reduce the size of the directory while in others they remain almost unchanged. This happens because Dir-S is not able to maintain the shared working set in all cases. In contrast, when true positives are maintained, false positives can be more numerous because a higher number of private blocks increase the number of elements that have to be tracked by *Dir-P*. Even in such extreme situations, where the false positives increase noticeably, counter saturation never occurs in any of the runs of our evaluations.

7.3. Adaptive Dir-S/Dir-P size ratio

So far the directory space available has been divided in equal parts between the two structures; *Dir-S and Dir-P*. However, Fig. 7 clearly indicates that multi-programmed and numerical workloads require a lower number of *Dir-S* entries while they would benefit from having a large *Dir-P* store. On the contrary, on-chip traffic of commercial workloads are dominated by the compulsory reconstructions. Therefore, although we did not implement the on-line adaptation, we can statically choose the best configuration for the application. For example, we could select a few-way *Dir-S* and several sub-tables for *Dir-P* for the first two classes and vice versa,

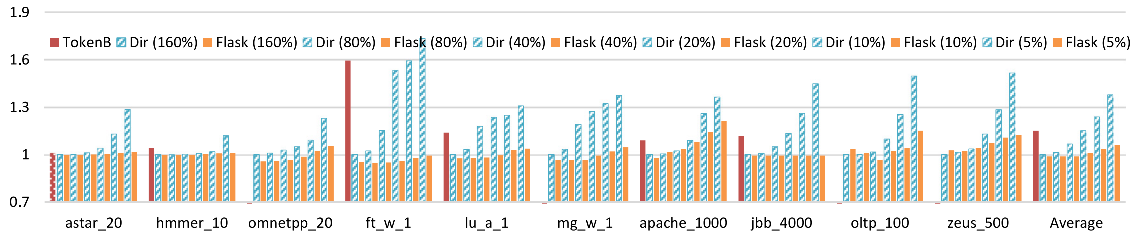

Fig. 4. Directory normalized execution time. Results have been normalized against the overprovisioned directory (160%).

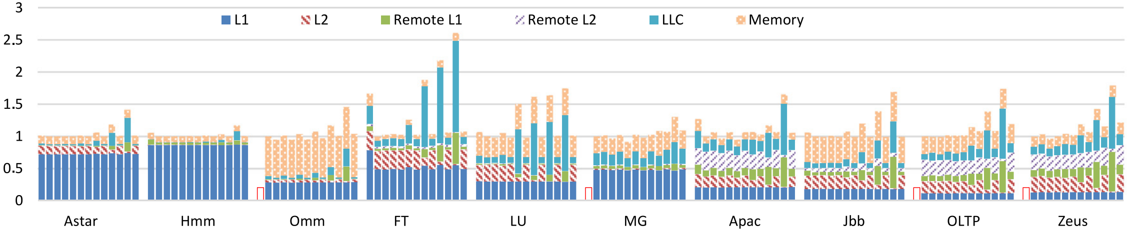

Fig. 5. Directory normalized average memory access time.

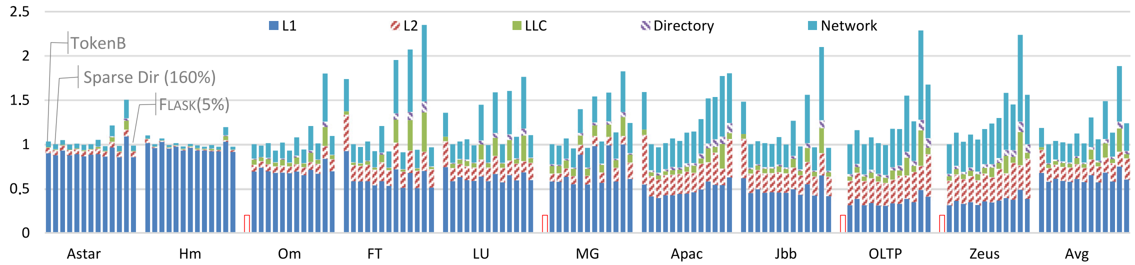

Fig. 6. Directory normalized on-chip memory hierarchy EDP.

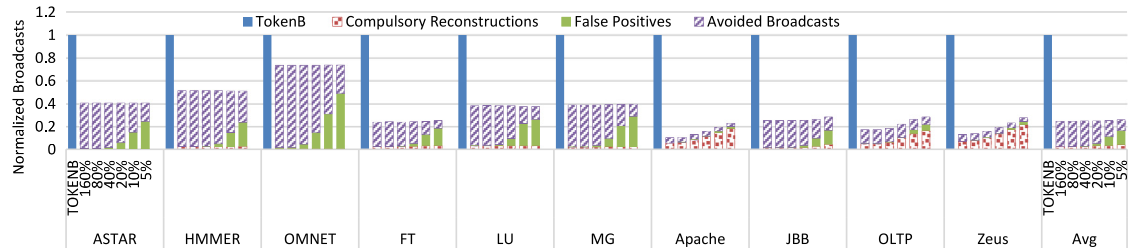

Fig. 7. TokenB normalized filter efficiency for different SDE sizes using ideal memory controllers.

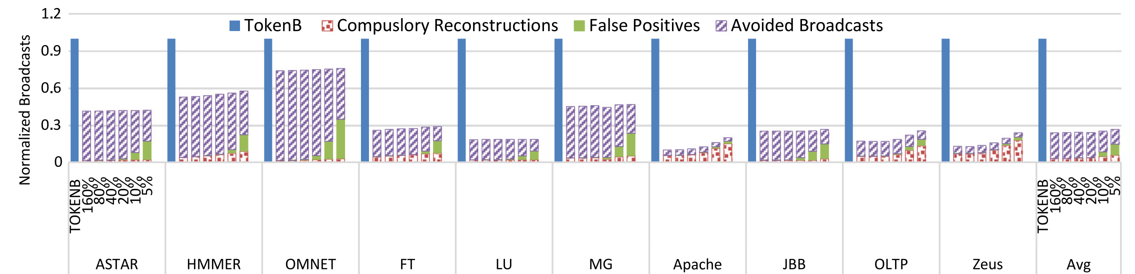

Fig. 8. TokenB normalized traffic filtering with adaptive partitioning.

i.e. several-way *Dir-S* for the last class with just a few sub-tables for *Dir-P*. Although the protocol works with zero size assigned to any of the two structures, it must be taken into account that a small number of *OS* addresses might be shared, so we need at least a 1-way in *Dir-S*.

Selecting the best *Dir-S*/*Dir-P* configuration for each application shows that it is possible to reduce the storage resources with negligible impact. This can be seen comparing Figs. 7 and 8. The former shows the number of broadcasts obtained with half of the space devoted to each structure and the latter employing the best configuration for each application. We see that the amount of false positives is practically halved when storage come down under 20% of SDE capacity. This is translated to performance results as is shown in Fig. 9. For example, the performance penalty with 10%

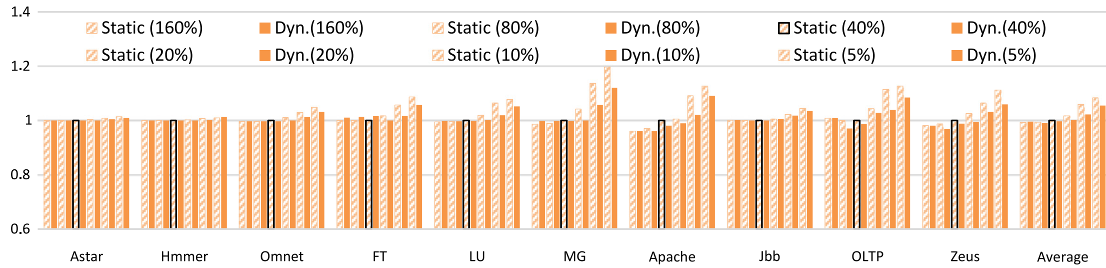

Fig. 9. 40% SDE static normalized execution time of FLASK with static and dynamic allocation.

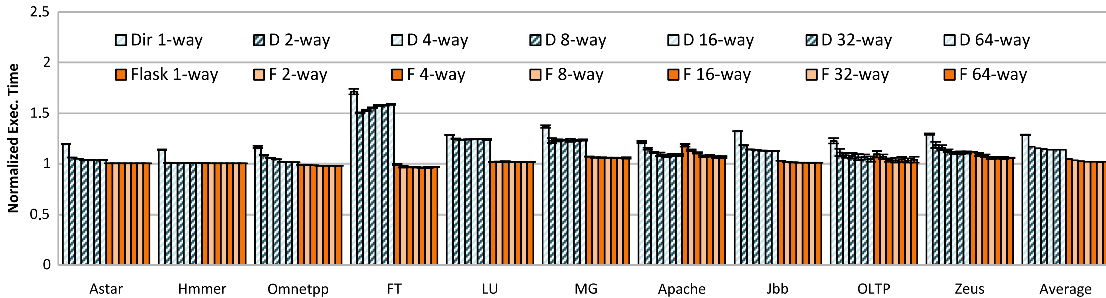

Fig. 10. 160% SDE sparse directory normalized performance for different associativities (20% SDE).

and 5% of SDE capacity is about 3% and 6%, respectively. This means that, under these conditions, with 10% SDE capacity the traffic requirements are close to those obtained with 20% of SDE statically halved.

These promising results suggest the search for online mechanisms capable of modifying the ratio of sizes *Dir-S*/*Dir-P* to adapt to the characteristics of the application. Although we will leave this analysis for future work, we believe that this adaptation should be performed in the software layer as the change of behavior is quite infrequent.

7.4. *Comparing* FLASK *with other directory cost reduction alternatives*

There are numerous alternatives which focus on the same goal as FLASK [10,11,13,27,35]. Many of them can be combined with our proposal to improve its performance. However, it is interesting to make a comparison between FLASK and those works which eliminate the directory overprovision by emulating large associativity through multi-hashing indexing and insertion [35,34]. The aim in all cases is to obtain similar values of application performance but with a lower cost of implementation. Therefore, the comparison can be established based on the impact caused by the limited associativity of the directory that ultimately is what causes conflict and loss of performance.

In Fig. 10, the execution time of the workloads has been represented varying the associative of the directory in a range from 1-way to 64-way with a fixed capacity of 20% SDE to appreciate the evictions caused by conflicts in the directory. The values have been normalized to the 160% SDE over-provisioned sparse directory. Except in the case of the FT workload, whose counterintuitive behavior is caused by very low directory reuse [35], increasing the associativity reduces the directory conflicts, which provides a slight performance degradation. Obviously, reduction to a 1-way directory is not realistic. However, the comparison is important because as you gradually increase the number of cores and the associativity of private caches, the number of blocks that map into the same directory entry (unless we want to quadratically increase its size) will increase.

It can be seen from Fig. 10 that 1-way directory in FLASK is able to outperform the sparse directory even with 64 ways. In FLASK, the impact of directory conflicts on performance is negligible. Therefore, our proposal outperforms techniques focused on minimizing directory conflicts such as [10,11,27].

This is because: (1) in FLASK, is not necessary to perform invalidations in private caches after a directory eviction, (2) only the actively shared blocks are tracked, (3) the need for inclusiveness is removed, and (4) both search times and bandwidth are reduced by using tokens and the tracking structure, *Dir-P*. This allows a significant reduction in storage devoted to directory, with little or no performance degradation.

8. Conclusions

This work introduces a proposal for a new coherence protocol for CMPs which is able to take advantage of both main types of protocol: snoop-based and directory-based mechanisms. The directory size can be reduced by dividing it into two different structures. One to track only the blocks that are being actively shared among the cores and the other to determine the presence of a block in the private caches. In our mechanism, inclusivity is avoided by using a token-based mechanism which is also helpful to simplify the correctness of the solution. The pressure on each of the structures is complementary and its assignment can be modified based on the sharing degree of the applications that will be run in the system. This also helps by further reducing the total storage space dedicated to each of them.

Exhaustive searches for the requested blocks through broadcasts are reduced by only using them when there is not enough space to assign to each of the structures and when a block state changes from private to shared. In both cases, these searches are limited to being done inside the chip in order to avoid saturating the off-chip bandwidth. The results enable a balanced approach that improves system performance in a wide range of applications.

The proposal might be beneficial to scale the coherence protocol for many-core systems or for medium-sized CMPs, as we have demonstrated in the results section; especially so bearing in mind recent and forthcoming commercial systems. In any case, we have shown that even for CMPs with sixteen aggressive cores there are both power and performance benefits versus other protocols.

Acknowledgments

This work was supported by the Spanish Secretary of State for R&D&i under contracts TIN2016-80512-R and TIN2015-66979-R (MINECO/FEDER, UE) and by the HiPEAC European Network of Excellence.

References

[1] P. Abad, P. Prieto, L.G. Menezo, A. Colaso, V. Puente, J.-Á. Gregorio, TOPAZ: An open-source interconnection network simulator for chip multiprocessors and supercomputers, in: International Symposium on Networks-on-Chip, NOCS, 2012, pp. 99–106.

[2] A.R. Alameldeen, M.M.K. Martin, C.J. Mauer, K.E. Moore, M.D. Hill, D.A. Wood, D.J. Sorin, Simulating a $2M commercial server on a $2K PC, Comput. (Long. Beach. Calif) 36 (2) (2003) 50–57.

[3] M. Alisafaee, Spatiotemporal coherence tracking, in: International Symposium on Microarchitecture, MICRO, 2012, pp. 341–350.

[4] E.E. Bilir, R.M. Dickson, Y. Hu, M. Plakal, D.J. Sorin, M.D. Hill, D.a. Wood, Multicast snooping: a new coherence method using a multicast address network, in: Proc. ISCA '99, Vol.0, no. c, 1999, pp. 294–304.

[5] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.

[6] F. Bonomi, M. Mitzenmacher, R. Panigrahy, An improved construction for counting bloom filters, in: Annual European Symposium, ESA, 2006, pp. 684–695.

[7] A. Broder, M. Mitzenmacher, Network applications of bloom filters: A survey, Internet Math. 1 (4) (2004) 485–509.

[8] M. Butler, AMD 'Bulldozer' Core - a new approach to multithreaded compute performance for maximum efficiency and throughput, in: Symposium on High-Performance Chips, HotChips, 2010.

[9] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, B. Hughes, Cache hierarchy and memory subsystem of the AMD opteron processor, IEEE Micro 30 (2) (2010) 16–29.

[10] B.A. Cuesta, A. Ros, M.E. Gómez, A. Robles, J.F. Duato, Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks, in: Int. Symposium on Computer Architecture, ISCA, 2011, p. 93.

[11] S. Demetriades, S. Cho, Stash directory: A scalable directory for many-core coherence, in: Int. Symp. on High Perf. Computer Architecture, HPCA, 2014.

[12] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area Web cache sharing protocol, IEEE/ACM Trans. Netw. 8 (3) (2000) 281–293.

[13] M. Ferdman, P. Lotfi-Kamran, K. Balet, B. Falsafi, Cuckoo directory: A scalable directory for many-core systems, in: International Symposium on High Performance Computer Architecture, HPCA, 2011, pp. 169–180.

[14] E.J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille, D. Plass, R. Puri, P. Restle, D. Shan, K. Stawiasz, Z.T. Deniz, D. Wendel, M. Ziegler, POWER8: A 12-core server-class processor in 22nm SOI with 7.6Tb/s off-chip bandwidth, in: International Solid-State Circuits Conference, ISSCC, 2014, pp. 96–97.

[15] A. Gupta, W. Weber, T. Mowry, Reducing memory and traffic requirements for scalable directory-based cache coherence schemes, in: International Conference on Parallel Processing, ICPP, 1990, pp. 312–321.

[16] P. Hammarlund, A.J. Martinez, 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.

[17] A. Jaleel, K.B. Theobald, S.C. Steely, J. Emer, High performance cache replacement using re-reference interval prediction, RRIP, in: International Symposium on Computer Architecture, ISCA, 2010, pp. 60–72.

[18] N.E. Jerger, L.S. Peh, M. Lipasti, Virtual circuit tree multicasting: A case for on-chip hardware multicast support, in: International Symposium on Computer Architecture, ISCA, 2008, pp. 229–240.

[19] H. Jin, M. Frumkin, J. Yan, The OpenMP implementation of NAS parallel benchmarks and its performance, Technical Report NAS-99-011, NASA Ames Research Center, Citeseer, 1999.

[20] P. Kongetira, K. Aingaran, K. Olukotun, Niagara: A 32-way multithreaded sparc processor, IEEE Micro 25 (2) (2005) 21–29.

[21] T. Krishna, L.-S. Peh, B.M. Beckmann, S.K. Reinhardt, Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication, in: International Symposium on Microarchitecture, MICRO, Vol. 2, 2011, pp. 71–80.

[22] A. Kumary, P. Kunduz, A.P. Singhx, L.-S. Pehy, N.K. Jhay, A 4.6Tbits/s 3.6GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS, in: International Conference on Computer Design, CCD, 2007, pp. 63–70.

[23] P. Lotfi-Kamran, M. Ferdman, D. Crisan, B. Faisafi, TurboTag: Lookup filtering to reduce coherence directory power, in: LowPower Electron. Des. ISLPED 2010 ACM IEEE Int. Symp., 2010, pp. 377–382.

[24] M.M.K. Martin, M.D. Hill, D.A. Wood, Token Coherence: decoupling performance and correctness, in: Int. Symp. on Computer Arch. ISCA, 2003, pp. 182–193.

[25] M.M.K. 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, ACM SIGARCH Comput. Archit. News 33 (4) (2005) 92.

[26] M.M.K. Martin, D.J. Sorin, M.D. Hill, D.A. Wood, Bandwidth adaptive snooping, in: International Symposium on High Performance Computer Architecture, HPCA, pp. 251–262.

[27] L.G. Menezo, V. Puente, J.A. Gregorio, The case for a scalable coherence protocol for complex on-chip cache hierarchies in many-core systems, in: International Conference on Parallel Architectures and Compilation Techniques, PACT, 2013, pp. 279–288.

[28] L.G. Menezo, V. Puente, J.-A. Gregorio, Flask coherence: A morphable hybrid coherence protocol to balance energy, performance and scalability, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA, 2015, pp. 198–209.

[29] A. Moshovos, RegionScout: Exploiting coarse grain sharing in snoop-based coherence, in: International Symposium on Computer Architecture, ISCA, 2005, pp. 234–245.

[30] N. Muralimanohar, R. Balasubramonian, N. Jouppi, Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0, in: International Symposium on Microarchitecture, MICRO, 2007, pp. 3–14.

[31] A. Raghavan, C. Blundell, M.M.K. Martin, Token tenure: PATCHing token counting using directory-based cache coherence, in: International Symposium on Microarchitecture, MICRO, 2008, pp. 47–58.

[32] M.V. Ramakrishna, E. Fu, E. Bahcekapili, Efficient hardware hashing functions for high performance computers, IEEE Trans. Comput. 46 (12) (1997) 1378–1381.

[33] A. Ros, M. Davari, S. Kaxiras, Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies Alberto, in: High Performance Computer Architecture, HPCA, 2015, pp. 186–197.

[34] D. Sanchez, C. Kozyrakis, The ZCache: Decoupling ways and associativity, in: International Symposium on Microarchitecture, MICRO, 2010, pp. 187–198.

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

[36] SLICC specification of Flask and Counterpart Coherence Protocols. [Online]. Available: http://www.atc.unican.es/galerna/flask.

[37] SPEC Standard Performance Evaluation Corporation, "SPEC 2006".

[38] C. Sun, C.-H.O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, V. Stojanovic, DSENT - A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling, in: International Symposium on Networks-on-Chip, NOCS, 2012, pp. 201–210.

[39] B. Vöcking, How asymmetry helps load balancing, J. ACM 50 (4) (2003) 568–589.

[40] J. Zebchuk, A. Moshovos, A tagless coherence directory, Comput. Eng. (2009) 423–434.

[41] H. Zhao, A. Shriraman, S. Dwarkadas, V. Srinivasan, SPATL: Honey, I shrunk the coherence directory, in: 20th International Conference on Parallel Architectures and Compilation Techniques, PACT'11, 2011, pp. 33–44.

Lucia G. Menezo received her B.S. and M.S. from the University of Basque Country in 2007. In 2014, she received her Ph.D. degree from the University of Cantabria, where she works as a researcher since then. Her research interests focus on the memory hierarchy, mainly on cache coherence protocols for chips multiprocessor (CMPs).

Valentin Puente received the B.S., M.S. and Ph.D. degrees from the University of Cantabria, Spain, in 1995 and 2000, respectively. He is currently an associate professor of computer architecture at the Department of Computers and Electronics of the same University. His research interests focus on memory hierarchy design and the impact that incoming technology changes might have on it.

Jose-Angel Gregorio received the B.S., M.S. and Ph.D. degrees in physics (electronics) from the University of Cantabria, Spain, in 1978 and 1983, respectively. He is currently a professor of computer architecture in the Department of Computers and Electronics in the same university. His main research interests focus on chip multiprocessors (CMPs) with special emphasis on the memory subsystem, interconnection network and coherence protocol of these systems.