Microprocessors and Microsystems 35 (2011) 695–707

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

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

Filtering directory lookups in CMPs

A. Bosque [a,*], V. Viñals [b], P. Ibáñez [b], J.M. Llabería [a]

[a] Department of Computer Architecture, Universitat Politécnica de Catalunya (UPC), Barcelona, Spain
[b] Department of Computer Science and Systems Engineering, University of Zaragoza, Zaragoza, Spain

ARTICLE INFO

Article history:
Available online 22 August 2011

Keywords:
Chip multiprocessor (CMP)
Coherence directory
Coherence actions filtering

ABSTRACT

Coherence protocols consume an important fraction of power to determine which coherence action to perform. Specifically, on CMPs with shared cache and directory-based coherence protocol implemented as a duplicate of local caches tags, we have observed that a big fraction of directory lookups cause a miss, because the block looked up is not allocated in any local cache. To reduce the number of directory lookups and therefore the power consumption, we propose to add a filter before the directory access.

We introduce two filter implementations. In the first one, filtering information is explicitly kept in the shared cache for every block. In the second one, filtering information is decoupled from the shared cache organization, so the filter size does not depend on the shared cache size.

We evaluate our filters in a CMP with 8 in-order processors with 4 threads each and a memory hierarchy with write-through local caches and a shared cache. We show that, for SPLASH2 benchmarks, the proposed filters reduce the number of directory lookups performed by 60% while power consumption is reduced by ~28%. For Specweb2005, the number of directory lookups performed is reduced by 68% (44%), while directory power consumption is reduced by 19% (9%) using the first (second) filter implementation.

1. Introduction

During the past decade single-core processors have become increasingly more complex, even leading to designs with inefficiencies in power/performance. This fact, along with the always increasing density in the chips, has encouraged the development of multi-core chips. Nowadays, most computer manufacturers offer multi-core chips such as the IBM Power 6 with two cores [1], the AMD Phenom II with four cores [2], the Fujitsu SPARC64 VII with four cores [3], the Intel Xeon 7400 series with six cores [4], or the SUN Niagara 2 with eight cores [5].

All these systems differ from each other in important features like the number of cores, the memory hierarchy, or the interconnection network on-chip. However, all of them support the shared memory programming model, so a coherence protocol is necessary to keep local caches coherent.

Coherence protocols can be classified as directory-based or snoopy-based protocols. Directory-based protocols keep a directory that stores the state of each block of main memory. All transactions should access this structure in order to determine which coherence actions to perform. In the snoopy-based protocols, the state of each block of cached data is stored in the local caches, that is, the information about the state of the cached data is distributed. As a result, all transactions are broadcasted to all the local caches in the system.

Both protocols consume an important fraction of shared cache energy to determine which action to perform. The energy in snoopy-based protocols is spent on broadcasting coherence messages and making tag-cache lookups [6]. Directory-based protocols consume less energy than snoopy-based protocols, because it is known which caches have a copy of a block [7].

A directory can be implemented in two basic ways: by a full-map scheme [8], or by duplicating the local cache tags [9]. Differences between duplicate tag directory and full-map arise in size, lookup method, and retrieved information in a lookup operation. Concerning size, the duplicate tag directory uses the smallest explicit representation of all blocks contained in local caches. Thus, a duplicate tag directory requires less area than a full-map directory. However, by duplicating local cache tags, any directory lookup requires an associative lookup that is expensive in terms of energy consumption. For example, in Niagara 2, a lookup can perform up to 256 comparisons. Both directory schemes identify the processors that have a copy. However, the duplicate tag directory scheme also identifies the way in the set of the local caches owning the copy and so there is no need of local cache lookups to identify which block to invalidate.

In directory-based protocols, we observe that an important fraction of directory lookups "misses", that is, the searched block is not cached in any local cache in the system. These directory lookups waste energy because we could decide not to perform them and

* Corresponding author. Tel.: +34 934017423; fax: +34 934017055.
E-mail addresses: abosque@ac.upc.edu (A. Bosque), victor@unizar.es (V. Viñals), imarin@unizar.es (P. Ibáñez), llaberia@ac.upc.edu (J.M. Llabería).

0141-9331/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2011.08.006

the program execution would be correct. We suggest to identify these useless lookups and avoid to perform them.

We propose two filter implementations that are accessed before the directory. These filters dismiss lookups that would "miss" in the directory. All proposed filters are designed for CMPs like Niagara 2 where the local cache level is split into an instruction cache and a data cache [5], the shared cache is inclusive, and the directory is implemented by duplicating the local cache tags.

Although instruction and data streams generally access different memory regions, we cannot guarantee that a cache line has been accessed just from one and only one of these streams (e.g., in self-modifying codes, the same block may be accessed as data and instruction). As a result, it is necessary to check both the copy of the tags of the local data caches and the copy of the tags of the local instruction caches in every search in a directory implemented as a copy of the local tags.

The local cache level in the evaluated CMP is split into data and instruction. Thus, every access to the shared cache might be labeled with the stream (data or instruction) it belongs to. The proposed filters use this information to decide, based on the previous accesses, if a block belongs to the data or instructions stream. When the filter is able to classify a block as exclusively belonging to a specific stream, it is only necessary to access the directory of that stream. All directory lookups performed in the other directory are filtered out.

Along this paper, we propose two basic different filter implementations. In the first one, we exploit the inclusion property of the shared cache to label each block with the stream it belongs to. In the second one, we keep the information of all blocks belonging to a stream together. We use Bloom filters that are space-efficient probabilistic data structures used to test whether an element is a member of a set [10]. As we need to identify two different streams, we use two independent Bloom filters. Each Bloom filter is associated with a directory and it keeps information of all blocks allocated in that directory.

The rest of this paper is organized as follows. We motivate our idea in Section 2. In Section 3 we outline the proposed filters. Section 4 describes in detail the memory hierarchy of the chosen CMP model. In Section 5 we detail the filter implementations. Section 6 shows our experimental results and Section 7 discusses related work. Finally, Section 8 contains the conclusions.

2. Motivation

We model a CMP with a shared inclusive L2 cache, multithreaded processors that access local instruction and data caches, and use write-through policy in local data caches. A detailed description of the CMP is in Section 4.

As the directory is implemented by duplicating the tags of the local cache, it is possible to distinguish between a data directory and an instruction directory. The data directory holds the tags of the local data caches whereas the instruction directory holds the tags of the local instruction caches.

In general, the sets of memory addresses of instructions and data (values of program counters of the executed instructions and addresses of referenced data) are disjoint. However, both stores and evictions need to look up both data and instruction directories because we cannot assure that instruction and data are always located in different memory regions. Two examples of this situation are self-modifying code and constants located in the code segment. The latter case happens when a compiler locates program constants along with the instructions that use them. Therefore, a cached block could contain instructions and constants, so there could be copies of that block in both the local data and instruction caches. If this block needs to be evicted from the shared cache, all the copies in the system have to be invalidated.

Instruction/data block exclusivity is maintained in the local caches, as in Niagara 2 [11]. In order to keep instruction/data block exclusivity, instruction fetches and loads that miss in the local caches also perform directory lookups.

Consequently, any access to the shared cache (loads that miss in the local data cache, instruction fetches that miss in the local instruction cache, and stores) and any eviction from the shared cache perform a directory lookup. Lookups are only useful if they hit, that is, if the target block is present in any local cache in the system. For the rest of the paper, we will call "useful lookups" to the lookup actions that hit in at least one local cache.

Fig. 1 shows the distribution of memory operations that access the shared cache and the total and useful directory lookups in the modeled CMP. Refer to Section 6 for a description of the used workloads and to Section 4 for the characteristics of the simulated system. Fig. 1 has three columns for each benchmark. The first one shows the memory operations categorized as load-misses, ifetch-misses, stores and evictions (bottom-up). The second column corresponds to the data and instruction directory lookups generated by the memory operations of the first column. The last column represents the number of useful data and instruction directory lookups.

The number of directory lookups is, on average, almost twice the number of memory operations. However, only 35% and 25% of the directory lookups are useful for SPLASH2 and Specweb2005, respectively. For SPLASH2, on average, the data directory lookups represent 40% of the directory lookups performed and 80% of them are useful. Thus, on average, 60% of the directory lookups are performed in the instruction directory and only 1% of them are useful. For Specweb2005, on average, the number of directory lookups are

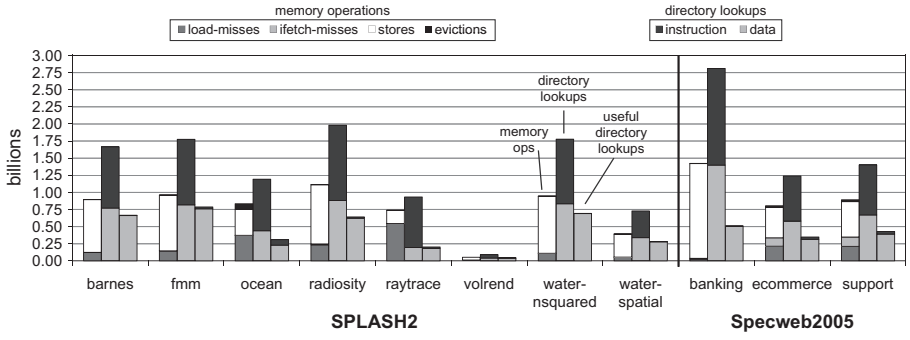

Fig. 1. The first column for each benchmark represents the billions of memory operations that access the shared cache categorized as load-misses, ifetch-misses, stores and evictions. The second column collects the billions of directory lookups in each directory. The third column represents the billions of "useful" directory lookups in each directory.

performed in equal number in both directories. However, while 50% of the data directory lookups are useful, only 2% of the instruction directory lookups are useful.

In a system where instruction/data exclusivity were not maintained, the number of directory lookups would be smaller. This difference can be computed from Fig. 1. The first column in Fig. 1 shows that, on average, 67% of the memory operations are stores for both SPLASH2 and Specweb2005, the number of ifetch-misses is almost zero for SPLASH2 and 6% of the memory operations for Specweb2005, and the number of load-misses is 22% and 10% of the memory operations for SPLASH2 and Specweb2005, respectively. Thus, the number of directory lookups necessary for maintaining instruction/data exclusivity represents a small fraction (22% and 16% on average for SPLASH2 and Specweb2005, respectively) of the total number of lookups.

Results from Fig. 1 clearly indicate that if we know in advance whether a directory lookup is useful or not, the number of performed lookups (and hence the energy consumption) can be greatly reduced.

3. Filtering mechanism outline

Based on the results in the previous section, we propose to implement a filter that is able to know if a block is in the data or the instruction directory. As a result, lookups are only performed in one of the directories (data or instruction), thus reducing the energy consumed.

We propose two different basic filter implementations. In the first one, filtering information is explicitly kept for every block in the shared cache. We exploit the inclusion property of the shared cache to label each block with information about the stream it belongs to (data or instruction). Thus, the filter is implemented as metadata associated with each block in the shared cache, similar to the state bits of the block. We call this kind of filter "instruction-data filter" (ID filter).

In the second basic implementation, filtering information is kept in structures decoupled from the shared cache organization. We use one structure for each stream (data or instruction). These structures keep information about all blocks belonging to the specific stream. We implement each structure with a Bloom filter. Bloom filters [10] are space-efficient probabilistic data structures used to test whether an element is a member of a set.

Both proposed filters guarantee by their implementation that they do not produce false negatives, that is, they never indicate that a block is not allocated in a directory when it is there. In the first implementation, the metadata associated with each block only identifies the stream the block belongs to when the block has belonged to the same stream along the whole execution. When the block has belonged to both streams, depending on the specific implementation, either the block is identified as unknown or the state of the system is modified to guarantee that the block belongs to a specific stream. This will be cover with more detail in Sections 5.1 and 5.2. In the second implementation, we use Bloom filters, which guarantee that they never produce false negatives. As a result, we guarantee that whenever the target block is allocated in the directory, the lookup is performed.

The proposed filters are read in parallel with the shared cache tag array. Then, if necessary, the directory is accessed in parallel with the shared cache data array as in Niagara 2.

4. Chip multiprocessor model

Fig. 2 shows the CMP configuration we assume in this work. It is a CMP with 8 in-order multithreaded cores and a memory hierarchy similar to the one in Niagara 2. The first cache level is local to each core, and is composed of an instruction cache (L1 I) and a write-through no-write-allocate data cache (L1 D). Each core has also a store buffer (SB) with several entries per thread that contain all outstanding stores. The second-level cache (L2) is shared among all the cores and is inclusive, that is, all data in the local caches must also be in the shared cache. It is divided into different banks interleaved by second-level cache block size. Two crossbars communicate the two cache levels.

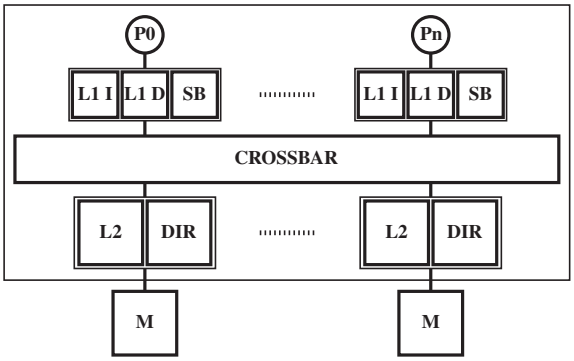

Fig. 2. CMP model with a first-level local cache per core (instruction cache, data cache and store buffer) and a second-level shared cache divided in several banks.

A write-invalidate directory-based protocol is used to maintain cache coherence among the local caches. The directory is distributed among the second-level cache banks, keeping close to each bank the information about the blocks associated with it. Table 1 presents the specific parameters we chose for the memory hierarchy. All of them are based on Niagara 2 memory hierarchy parameters.

CMPs that use write-through local caches (as the one modeled in this paper) require more bandwidth than CMPs that use write-back local caches (like Piranha [12]), because all stores must access the shared cache. However, the extra bandwidth guarantees that data is always updated in the shared cache. Thus, the latency to access shared data does not depend on how many caches are sharing it, but it is only increased by contention in the interconnection network. Moreover, in a system with write-back local caches, local caches are responsible for serving dirty blocks when they are requested by other processors. These requests can delay the access of a processor to its local caches, reducing processor's performance. In a system with write-through local caches, local caches do not serve any other processor requests.[1]

We assume a directory similar to that of Niagara 2 [11], which consists of a copy of the local cache tags. The directory is split into instruction and data directories, replicating the organization of the local caches. The directory in each bank is implemented as a CAM structure whose area requirements are `O(PxNL1/NBL2)`, being `P` the number of processors, `NL1` the number of lines in the local caches and `NBL2` the number of banks of the shared cache. The size of this structure also depends on the local cache tag size. As the shared cache is inclusive, any local cache block is allocated in the shared cache. Thus, the local cache block tag in the directory could be replaced by the set index and way of the corresponding shared cache block. Since in the modeled CMP the shared cache tag array is accessed before the directory, set index and way are available on time to access the directory. This information requires fewer bits, and so, the directory size and its power consumption are smaller.

[1] The bandwidth requirements to implement local write-through caches is so high in many-core systems (which is not the target of this paper), that makes it unaffordable. However, in this scenario, a practical implementation would be to organize the many-core system in small clusters and perform cache coherency at the cluster level. Each of these clusters would be the CMP modeled in this paper.

Table 1
Memory hierarchy parameters.

L1 D size	8 KB	L2 size	4 MB
L1 D associativity	4-Way	L2 number of banks	8
L1 D block size	16B	L2 associativity	16-Way
L1 I size	16 KB	L2 block size	64B
L1 I associativity	8-Way	L2 latency	7 Cycles
L1 I block size	32B	L2 MSHR	8
Crossbar arbitration	3 Cycles	Store Buffer	8 Entries per thread
Crossbar latency	3 Cycles		
Physical address	40 Bits	Memory latency	117 Cycles

A lookup in a duplicate tag directory is associative, so it is expensive in terms of energy consumption. However, this lookup identifies not only the processors that have a copy, but also the way in the set of the local caches. Block invalidations are performed by sending to all involved processors a message that includes the local cache set index and the way (s) in the set to invalidate. So, there is no need of local cache lookups to identify which block to invalidate. The directory is also responsible for identifying which stores have to update a block in the local cache of the processor performing the store. Thus, stores update local caches when the acknowledgement message is received. This message, like invalidation messages, includes the way that has to be updated.

Like in Niagara 2 [11], instruction/data block exclusivity is maintained in the local caches, that is, the same block can not be at once in both instruction and data caches (across all cores). The directory is responsible for ensuring instruction/data exclusivity. The shared cache block size is larger than the block size of the local caches. Thus, copies of different subblocks from the same shared cache block can reside in local caches of different types (instruction/data).

4.1. Directory organization in a bank

Duplicate instruction and data directories have a similar structure, but they are accessed in a different way depending on the kind of memory operation.

In order to understand better the directory organization, let us first assume that the shared cache block size is the same as the local cache block size. The shared cache is split in several banks and it is interleaved by its block size. The directory is also split in order that each shared cache bank only keeps the fragment of the directory corresponding to the blocks mapped to that bank. Blocks located in contiguous local cache sets are mapped to different shared cache banks and therefore to different directories. Thus, the number of local cache blocks assigned to a particular directory is `(NLCS/SCB)*LC`, being `NSLC` the number of local cache sets, `SCB` the number of shared cache banks, and `LC` the number of local caches. Fig. 3 shows an example of how the blocks in the local cache sets are mapped to the different directories. We can see that blocks located in the first set of a local cache are mapped to the directory of the first shared cache bank, blocks located in the second set are mapped to the directory of the second shared cache bank, and so on. Blocks located in a specific set are mapped to a single directory, independently of the local cache where they are located.

However, as the local cache block size is smaller than the shared cache block size in our CMP model, several contiguous local cache sets are mapped to the same shared cache bank. This number is equal to the ratio between the shared cache and the local cache block size. As before, the number of blocks mapped to a particular directory is `(NLCS/SCB)*LC`. Fig. 4 shows the same system as Fig. 3, but now the local cache block size is half the shared cache

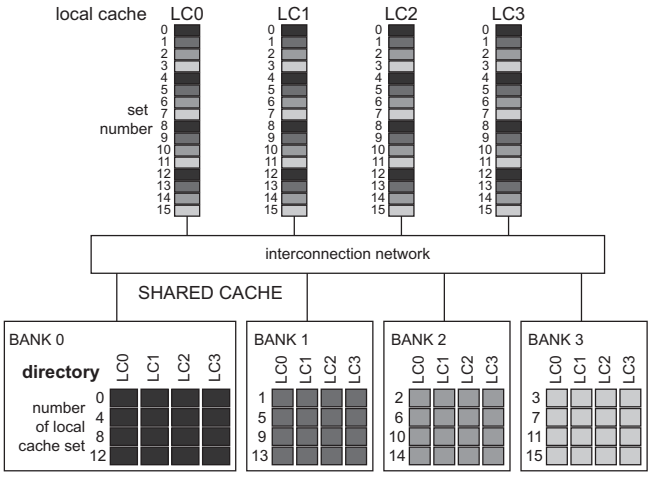

Fig. 3. Mapping of local cache blocks to shared cache banks and directories when the shared cache and the local caches have the same block size.

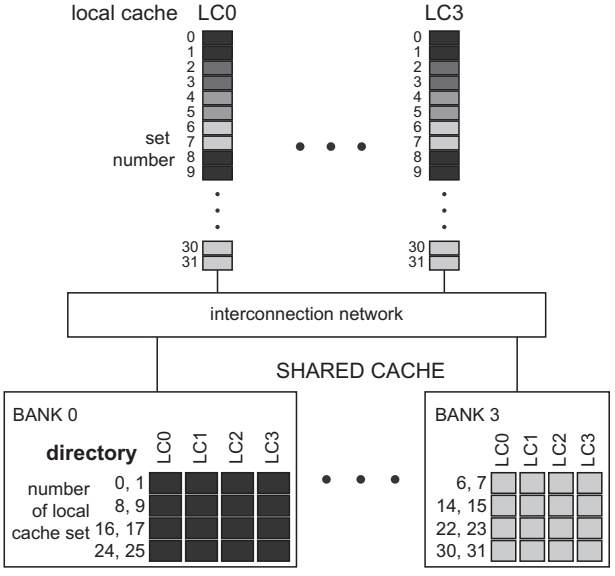

Fig. 4. Mapping of local cache blocks to shared cache banks and directories when the shared cache block size is twice the local cache block size.

block size. The number of sets in the local cache is doubled and the mapping to the shared cache banks changes. Blocks located in two consecutive local cache sets are mapped to the same shared cache bank: the first two local sets are mapped to the first shared cache bank, the second two local sets are mapped to the second shared cache bank, and so on.

Our CMP model has 8 shared cache banks interleaved by 64B blocks, and 8 local data caches, each one with 128 sets of associativity 4, and block size of 16B (see Table 1). Blocks located in four contiguous local cache sets are mapped to the same directory since the banks are interleaved by 64B blocks and the local cache block size is 16B. The total number of blocks mapped in a specific directory is 512 (4-way $\times$ 16 sets $\times$ 8 local caches).

Each directory puts together blocks located in the same set of all the local caches because it is the minimum amount of blocks that need to be looked up in any directory lookup. This number of blocks is 32 (4 blocks/set $\times$ 8 local caches). Moreover, the directory is also organized in order to make easy to access blocks located in contiguous sets in the local caches. The reason is that depending on the shared cache access, the corresponding directory lookup can

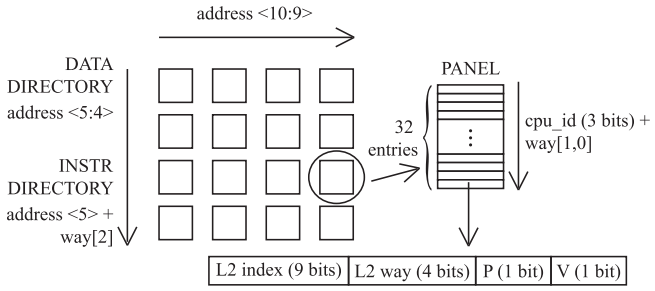

Fig. 5. Data or instruction directory structure of Niagara 2 and how they are accessed.

require to look up all these blocks. Fig. 5 shows how the directory is organized in 16 different panels (using SUN's terminology [11]) of 32 entries which correspond to the blocks located in the same set of all the local data caches. Each panel entry keeps only the block located in a particular local cache and in a specific way of the local cache set that the panel corresponds to. Any directory lookup determines the panel (s) which has to be looked up using some address bits of the target block. Then, all the entries of the panel (s) are compared to the target block. The local caches that keep a local copy and the way where the copy is located are determined by the entries with a positive result from the comparison.

In a similar way, the instruction directory of a shared cache bank tracks 512 blocks (8-way × 8 sets × 8 local caches). It is also organized in panels of the same size as the data directory. Differences are due to larger cache block size and higher associativity in the local instruction caches.

4.2. Directory operation

Any access to and any eviction from the shared cache performs a lookup in the directory. Along this paper, we call "memory operations" to all these accesses and evictions, namely, loads and instruction fetches that miss in the local caches, stores and evictions. If the shared cache and the local cache had the same block size, the directory lookup for all memory operations would require the same number of comparisons, involving all the elements of a specific local cache set in every local cache. However, in our CMP model local and shared cache block sizes are different. Thus, some lookups require twice or four times more comparisons.

Below we describe the memory operations and the actions performed in the directory for each one:

- *Load-miss*: It is a load that miss in the local data cache. The directory entry that corresponds to the local cache location in which the block will be allocated is updated with the missing address tag. In order to assure instruction/data exclusivity, it is necessary to invalidate all the copies of the 16B block (local data cache block size) in all the local instruction caches. The address of this 16B block determines a specific set in the local instruction caches. As there are 8 local instruction 8-way associative caches, 64 comparisons are performed.
- *Ifetch-miss*: It is an instruction fetch miss in the local instruction cache. The behavior is the same as in a load but, instead of the data directory, the instruction directory is updated and all the copies of the 32B block (local instruction cache block size) are invalidated in the local data caches. Thus, the blocks allocated in two local data cache sets are looked up. As there are 8 local data 4-way associative caches, 64 comparisons are performed.
- *Store*: As the local data cache is write-through, every store accesses the shared cache. The copy of the local tags in both directories, data and instruction, are looked up in order to send invalidations to all the local caches that have the block. The address of this block determines one local data cache set and one local instruction cache set. Thus, 32 and 64 comparisons are performed in the data and the instruction directories respectively.
- *Eviction*: As inclusion is enforced in the system, the shared cache victim block is removed from the local caches. Instruction and data directories are looked up in order to send invalidations to all the local caches that have a copy of the 64B evicted block (shared cache block size). Thus, up to two 32B blocks in the local instruction caches and four 16B blocks in the local data caches can be invalidated. So, 128 comparisons are performed in each directory.

Table 2
Actions performed in the data and instruction directories for every memory operation in the shared cache. For each lookup, the number of comparisons performed is enclosed. The shaded cells identify the actions that are unnecessary in a system without data/instruction exclusivity.

Memory operation	Data directory	Instruction directory
Load-miss	Update	**Lookup (64)**
Ifetch-miss	**Lookup (64)**	Update
Store	Lookup (32)	Lookup (64)
Shared cache eviction	Lookup (128)	Lookup (128)

Table 2 summarizes the actions performed for every memory operation and the number of comparisons performed by the lookup actions. The bolded cells identify the actions that are unnecessary in a system without data/instruction exclusivity.

Update actions are mandatory in order to keep always an exact copy of the tags of the local caches in the directory.

5. Filter implementation

In this section we first present two ID filter implementations where filtering information is associated with the shared cache block. The filter size is related with the shared cache size. Secondly, we present a Bloom based filter design. This filter is decoupled from shared cache bank organization and so, it is independent of the shared cache size.

5.1. A simple implementation: the two-bit ID filter

The two-bit ID filter classifies the shared cache blocks as belonging either to the data stream, instruction stream or both: blocks that have been accessed only by loads and stores are marked as *data blocks*; blocks that have been accessed only by instruction fetches are marked as *instruction* blocks; and blocks accessed by instruction fetches and stores or loads are marked as *mixed* blocks. Therefore, the filter contains two bits per shared cache block.

The value of these bits is set every time that a new block is allocated in the shared cache and it is updated only when the type of the block changes. A data (instruction) block changes its type when it is accessed by an ifetch-miss (load-miss or store). The type of the block changes to mixed in both cases.

Table 3 collects the actions performed in the directory depending on the memory operation and the type of the accessed block. Comparing Tables 2 and 3, we observe the following differences. For load-misses and ifetch-misses, the lookup actions can be eliminated for data and instruction blocks, respectively. For stores and evictions, as long as a block is classified as instruction or data, it is only necessary to look up in one directory. For a data (instruction) block, all its copies must be in the local data (instruction) caches, so only the data (instruction) directory should be looked up.

Table 3
Actions performed in the data and instruction directories when using the two-bit ID filter. The number of comparisons performed by each lookup is enclosed.

Memory operation	Block type	Data directory	Instruction directory
Load-miss	Data	Update	–
	Instr	Update	Lookup (64)
	Mixed	Update	Lookup (64)
Ifetch-miss	Data	Lookup (64)	Update
	Instr	–	Update
	Mixed	Lookup (64)	Update
Store	Data	Lookup (32)	–
	Instr	–	Lookup (64)
	Mixed	Lookup (32)	Lookup (64)
Shared cache eviction	Data	Lookup (4)	–
	Instr	–	Lookup (4)
	Mixed	Lookup (4)	Lookup (4)

This filter implementation has two drawbacks: it requires two bits per shared cache block, and its performance could be reduced if highly accessed blocks are classified as mixed. A block classified as mixed, as long as it remains in the shared cache, can not change its type. The reason is that the filter itself does not keep information about the number of copies of any block in the data and instruction local caches. So, after any access to the shared cache, the filter has not enough information to revert the state of a block from mixed to instruction or data. This information is available in the directory but, as the shared and the local caches block sizes are different, neither of the directory accesses looks up enough information to decide whether a block can be classified as data or instruction. So, for example, for an instruction block that is accessed as data once, even though hundreds of instruction fetches are performed over it, it will not be considered an instruction block anymore. Below we propose a different ID filter implementation to eliminate mixed blocks and to reduce the filter size.

5.2. A smaller filter: the one-bit improved ID filter

As blocks in the shared cache barely change their type, we propose an ID filter that classifies every block in the shared cache either as data or as instruction. This ID filter requires only one bit per shared cache block, so the filter size is halved.

For a proper operation of the one-bit ID filter, it is necessary to modify the coherence protocol to force instruction/data exclusivity of 64B blocks (shared cache block size). So, each block in the shared cache is forced to be classified either as a data or instruction block. Thus, there can be copies of a block either in the local data caches or in the local instruction caches, but never in both of them.

A drawback of the one-bit ID filter is that every time a block in the shared cache changes its type, all the copies of this block in the local caches must be invalidated. A block that changes its type from data to instruction (or instruction to data) needs to invalidate all its copies in the local data (or instruction) caches. In order to carry out the invalidations, a directory lookup is needed.

In general, changes in the type of block are rare in the analyzed workloads. However, in some specific workloads, a few amount of blocks highly accessed change continuously their type because they contain both instructions and data. These blocks come from the compiler placement of program constants in the code region. For example, the first 32B of a 64B block can hold instructions and the next 32B can hold data. In the system without filter, there are copies of the first 32B in the local instruction caches and copies of the next 32B in the local data caches. Therefore, both loads and instruction fetches that access this block hit in the local caches and do not need to access the shared cache. However, with the one-bit ID filter, instruction/data exclusivity at 64B blocks is forced. Thus, any ifetch-miss (or load-miss) that accesses the first (or last) 32B sets the block type to instruction (or data) and invalidates all the copies of the last (or first) 32B in the local data (or instruction) caches. As a result, the number of load-misses and ifetch-misses to the shared cache increases. Therefore, there is an increase in the number of directory lookups (recall Table 2) and all of them are useful since there are local copies of the block.

Table 4
Actions performed in the directories after looking up in the one-bit improved ID filter.

Memory operation	Block type	Data directory	Instruction directory
Load-miss	Data	Update	–
	Instr	–	–
Ifetch-miss	Data	Lookup (128)	Update
	Instr	–	Update
Store	Data	Lookup (32)	–
	Instr	–	Lookup (128)
Shared cache eviction	Data	Lookup (128)	–
	Instr	–	Lookup (128)

We propose the one-bit improved ID filter in order to reduce this waste of energy. This filter assures instruction/data exclusivity by preventing a block classified as instruction block to change its type. Thus, when a load accesses a block classified as instruction block, the data is supplied to the local data cache, but it is not allocated in the local data cache. In this way, the number of loads in the shared cache increases as with the one-bit ID filter, but neither directory lookups nor invalidations are necessary. Moreover, the number of ifetch-misses is the same as in the system without filter.

Table 4 shows the actions performed in the directory for each memory operation when using the one-bit improved ID filter. Comparing Tables 3 and 4, we see that, in general, forcing data/exclusivity for 64B blocks causes a bigger number comparisons when the memory operation changes the type of the block: when an ifetch-miss access a data block. But, as the mixed block type does not exist, stores and evictions either access the data directory or the instruction directory, but they never look up both as before.

The filtering information, like in the two-bit ID filter, is set in the allocation of new blocks in the shared cache and it is updated every time a block changes its type. Every time a new block is allocated in the shared cache, the associated filter bit is set to instruction or data depending on the current memory operation. When any block changes its type, the associated filter bit is updated to the new value.

5.3. Bloom based filter

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set [10]. False positives are possible, but false negatives are not. We use Counting Bloom filters [13] in order to be able to remove elements from the set.

We use a Bloom filter for the data directory and another one for the instruction directory. Each of these filters keeps track of the block addresses allocated in each directory. To do membership tests or updates in the filters, we use the lower bits of the block index in the shared cache.

Before performing a directory lookup the corresponding Bloom filter is accessed. If the membership test results negative, the directory lookup is useless. This lookup is not performed and we know that neither invalidations nor updates are necessary in the local caches. In case the test membership is positive, we cannot assure whether the directory lookup is useless or not, and hence it is performed.

The Bloom filter is updated in every directory update. When a load-miss or an ifetch-miss access the shared cache, a new block

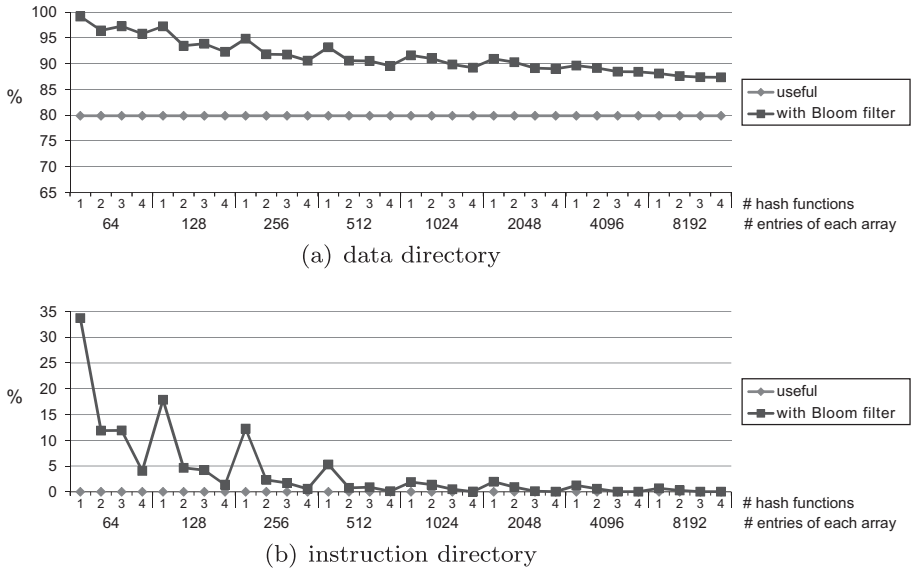

Fig. 6. Percentage of useful directory lookups and useful directory lookups according to the Bloom filter used for SPLASH2.

is allocated and another one is evicted in the corresponding local cache. The new block is inserted in the Bloom filter and the evicted block is removed. The insertion of the new block in the Bloom filter is performed using the block index in the shared cache of the block performing the load-miss or the ifetch-miss. In order to remove the evicted block, the directory has to be read since it keeps the shared cache indexes of the blocks currently allocated in the local caches.

The Bloom filter is also updated with local cache invalidations performed by stores, evictions, ifetch-misses and load-misses. Every block invalidated in the local caches has to be removed from the Bloom filter. The shared cache indexes of the invalidated blocks are known after a directory lookup.

We design Bloom filters of different sizes and using different number of hash functions in order to analyze their effectiveness. We choose two types of hash functions: based on bit shifts and based on XOR (exclusive or) operation. The hash functions based on bit shifts shift the address bits of the block accessing the filter a certain amount of bits and use the lower bits to access the Bloom filter. The hash functions that use an XOR operation split in half the address of the block accessing the filter, perform an XOR operation, and use the lower bits to access the Bloom filter. Based on bit shifts, we use three different hash functions: shift zero bits (`s0`), shift three bits (`s3`), and shift five bits (`s5`). Based on XOR operation, we use only one hash function (`xor`).

Fig. 6 shows the percentage of directory lookups that different Bloom based filters identify as useful for the data and the instruction directories for SPLASH2 (`with Bloom filter` in Fig. 6). The number of directory lookups that are useful is also included (`useful` in Fig. 6). The number of entries of the Bloom filter array vary from 32 to 8192. The number of hash functions varies between 1 and 4. There is an array of counters for each hash function and all of them are accessed in parallel. An increase in the number of hash functions involves an increase in the size of the Bloom based filter used. For example, when we use a 32-entry Bloom filter with 2 hash functions, 2 arrays of 32 entries each are used and each array is accessed by just one hash function. Table 5 shows which hash functions are used when we indicate 1, 2, 3, or 4 in Fig. 6.

Fig. 6a shows that the Bloom based filter of the data directory is able to identify very few useless directory lookups, even using Bloom filters with a big number of entries. For the instruction directory, the effectiveness of the Bloom based filter is good using quite small number of entries with enough number of hash functions.

Table 5
Different combination of hash functions used to analyze the effectiveness of the Bloom based filter.

Number of hash functions	Hash functions
1	xor
2	s3, s5
3	xor, s0, s3
4	xor, s0, s3, s5

Fig. 7 shows the same information as Fig. 6b but taking into account the size of the Bloom filter used. We can see that for the same filter size not all the filters behave in the same way. In general, the bigger the number of hash functions used, more directory lookups are identified as useless, that is, the number of directory lookups performed is reduced. Finally, we decide to use a Bloom filter with 128 entries and 2 hash functions since the reduction in the number of directory lookups identified as useful with bigger filters does not pay off the higher energy consumption of those filters. The counter size is 10 bits because each directory has 512 entries so 9 bit counters and a valid bit are needed. Thus, each Bloom filter has a size of 320B.

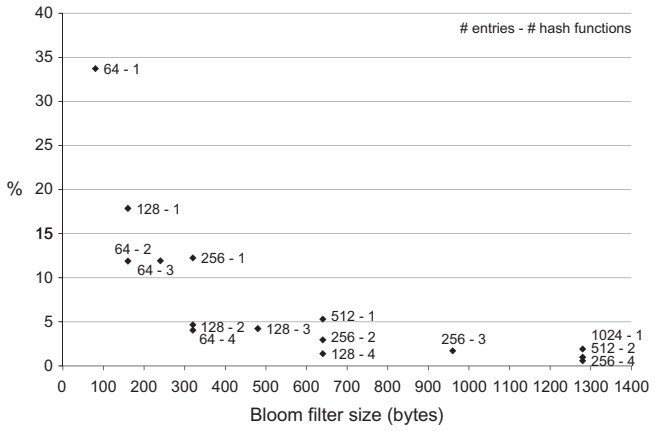

Fig. 7. Percentage of useful instruction directory lookups when using Bloom filters of different sizes for the instruction directory. Labels in the graph mean (# entries – # hash functions).

6. Evaluation

This section shows the benefits of the proposed filters. Section 6.1 describes the evaluation methodology. Section 6.2 shows the number of directory lookups performed when using the proposed filters, that is, the filter coverage. Section 6.3 analyzes the performance loss of the one-bit improved ID filter since it is the only proposed mechanism that changes the coherence protocol. Section 6.4 shows the energy saving when using each of the proposed filters. Finally, Section 6.5 shows the same results as Section 6.4 but for different local and shared cache sizes and different technologies.

6.1. Methodology

We use a Simics-based simulator. Simics [14] is a full-system multiprocessor simulator capable of running unmodified commercial OSs and applications. We completely developed the memory hierarchy of our CMP model on top of Simics using the tools provided by this simulator. The main characteristics of this memory hierarchy are detailed in Section 4 and its parameters are described in Table 1. The consistency memory model of this memory hierarchy is Total Store Order (TSO). We configured Simics to model a SPARC V9 target system running Solaris 9. We simulated a system with 8 in-order, blocking, 1.2 GHz processors with 4 threads each that share a 8 GB memory. Due to simulation time restrictions, the non-numerical applications are executed in a system with 8 non-multithreaded processors.

We use the applications of the SPLASH2 benchmark suite [15] and as non-numerical applications, the three workloads of Specweb2005 [16]: `banking`, `ecommerce`, and `support`. In order to adapt the SPLASH2 workloads to our simulated scenario, we scaled the input dataset up as proposed by Monchiero et al. [17]. For water-nsquared and water-spatial we were only able to scale the datasets to 2 k and 4 k particles, respectively to bound the simulation time. We execute the whole parallel section of each benchmark. Table 6 shows the applications used, the corresponding datasets and the billions of executed instructions.

For the three workloads of Specweb2005, we use the web server Apache 2.0.63 with PHP. Table 7 indicates how many simultaneous sessions are running for each workload. Web servers present high time and space variability [18] and, on top of that, we can not simulate Specweb2005 workloads until their completion due to simulation time restrictions. To minimize web server variability, we run several simulations for each workload performing the same number of web transactions, with a thinking time set to zero. All these simulations are run after fast forwarding the period of ramp up of Specweb2005 [16]. To determine the number of web transactions in each workload, we warm the caches for 0.75 billion cycles and then we measure the number of web transactions for 2.25 billion cycles. Table 7 shows the number of web transactions performed, the average billions of instructions executed for each workload, and the number of simulation runs. For Specweb2005 results we use the mean of the simulations.

Table 6
SPLASH2 benchmarks, the corresponding datasets, and billions of cycles and instructions executed.

Benchmark	Dataset	Instr (10^9)	Cycles (10^9)
Barnes	64 K particles	4.97	0.62
fmm	64 K particles	9.57	1.20
Ocean	1026x1026	5.99	0.91
Radiosity	– Largeroom, – ae 5000 – en 0.050 –bf 0.1	7.45	0.94
Raytrace	Balls4	5.77	0.79
Volrend	Head	0.63	0.08
Water-nsquared	2192 Particles	13.79	1.72
Water-spatial	4096 Particles	4.02	0.50

Table 7
Specweb2005 workloads, the corresponding simultaneous sessions, the number of web transactions, billions of instructions executed and number of simulation runs.

Workload	Simultaneous sessions	Web trans.	Instr (10^9)	Simulation runs
Banking	200	100	15.52	30
Ecommerce	1000	1200	8.07	15
Support	1400	2200	8.07	10

6.2. Coverage

In Fig. 1, we showed that a large amount of directory lookups are useless. Now, we analyze whether the proposed filters are able to identify the useless lookups in advance or not.

Fig. 8 shows the percentage of directory lookups identified as useful by each filter proposed and the percentage of useful directory lookups. The directory lookups identified as useful are those that will be performed, so the smaller the number of directory lookups classified as useful, the better. Fig. 8a and b show the percentage of useful instruction and data directory lookups, respectively. There are four columns per benchmark. From left to right, the first three correspond to the two-bit ID filter, the one-bit improved ID filter and the Bloom filter. The last column represents the percentage of useful directory lookups, which we call "perfect filter".

For both SPLASH2 and Specweb2005, Fig. 8a shows that less than 1% of the instruction directory lookups are useful. The two-bit and the one-bit improved ID filters identify almost all these cases. Thus, when using any of these filters, instruction directory lookups are reduced to less than 1% on average, compared to the system without filtering. The Bloom filter does not achieve such good results. For SPLASH2, the Bloom filter reduces instruction directory lookups to 4%, but for Specweb2005, it only reduces instruction directory lookups to 31%.

Bloom filter has a bad performance for Specweb2005 because its parameters are tuned for SPLASH2. We do not dedicate any effort to tune the Bloom filter parameters for Specweb2005 because later we show that, from a power consumption point of view, the performance of the Bloom filter for SPLASH2 is not better than the performance of any of the ID filters. Moreover, tunning the Bloom filter parameters for Specweb2005 requires to use more address bits in the hash functions. To get these extra bits the tag array should be read and this will increase the energy consumption.

Fig. 8b shows that the percentage of useful data directory lookups is far bigger than the percentage of useful instruction directory lookups. The reason is that stores represent an important fraction of the memory operations in the shared cache because local data caches are write-through (Fig. 1). Thus, most directory lookups performed in both directories are performed by stores which are accessing private data. This private data is located in the local data caches. As a result, instruction directory lookups are useless and can be safely filtered out. As stated before, the directory is used not only to find which blocks to invalidate in the local caches but also to identify the way in the set of the private cache that has to be updated with the store. So, data directory lookups generated by stores are, in general, useful and they should not be filtered out.

Fig. 8b shows that all SPLASH2 benchmarks except `ocean` behave similarly: on average, 85% of the data directory lookups are useful and none of the proposed filters is able to properly identify the useless directory lookups. The percentage of useless data directory lookups is less than 1% using any of the proposed filters compared to the system without filtering. The behavior of `ocean` is different: 50% of the data directory lookups are useless and both

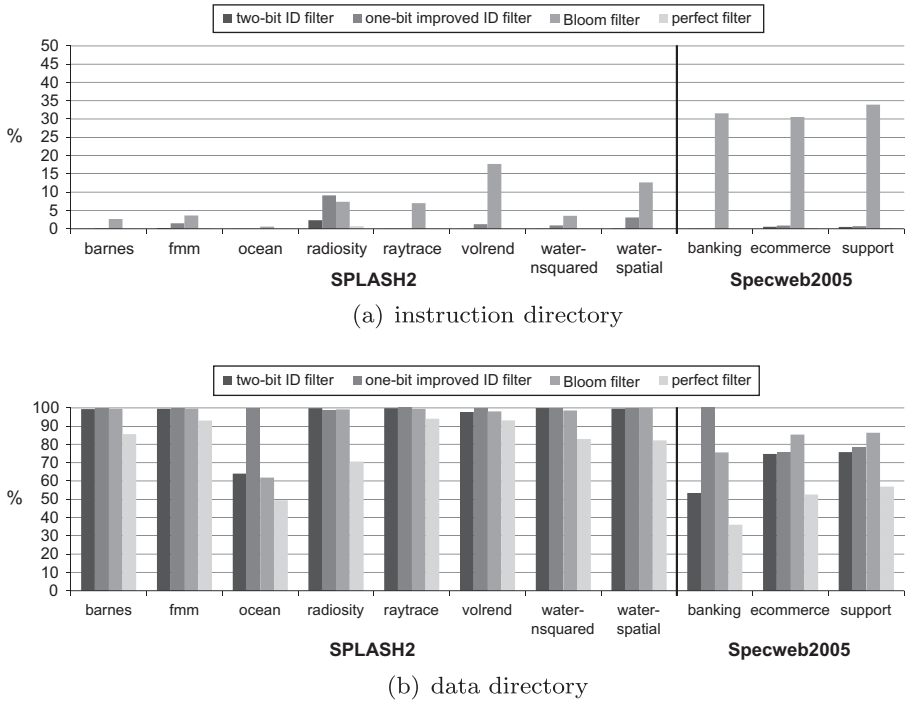

Fig. 8. Percentage of directory lookups identified as useful by each proposed filter and percentage of useful directory lookups. (a) Corresponds to the instruction directory lookups and (b) corresponds to the data directory. In both graphs, for every benchmark each column correspond to the system using a different filter: two-bit ID filter, one-bit improved ID filter, Bloom filter and a perfect filter (from left to right).

the two-bit ID filter and the Bloom filter are able to identify most of them, reducing the number of lookups up to 64% and 62%, respectively. `Ocean` differs from the rest of SPLASH2 benchmarks in that the number of shared cache evictions is as important as the number of stores. Most of the data directory lookups performed by evictions are useless and the proposed filters can identify them.

For Specweb2005, Fig. 8b shows that, on average, 50% of data directory lookups are useful, because an important number of data directory lookups are performed by ifetch-misses. Lookups performed by ifetch-misses are, in general, useless and can be filtered out. However, the proposed filters do not perform as well as in the instruction directory. The two-bit ID filter is the best, but it is only able to reduce the number of data directory lookups up to 67% compared to the system without filtering.

Summing up, all filters reduce the instruction directory lookups between 69% and 99%. However, data directory lookups are barely reduced. In the best case, for Specweb2005, they are reduced to 67%. The Bloom filter requires a specific filter for each directory, that is, a Bloom filter to filter out data directory lookups and another one to filter out instruction directory lookups. We propose to not use the Bloom filter in the data directory due to its poor performance. From now on, when we use a Bloom filter, we will be using a Bloom filter only for the instruction directory, that is, a Bloom filter will only filter out the instruction directory lookups while the data directory lookups will not be filtered out.

6.3. Performance

Both the two-bit ID filter and the Bloom filter do not modify the coherence protocol so benchmarks' performance is not altered. On the contrary, the one-bit improved ID filter modifies the coherence protocol forcing the instruction/data exclusivity at a 64B granularity. It is then necessary to check that the performance of the benchmarks is the same as before.

Fig. 9 shows the normalized execution time of the one-bit improved ID filter with regard to the system without any filter. For Specweb2005, it is interesting to compare both the mean and the standard deviation of the different simulations that we run (see

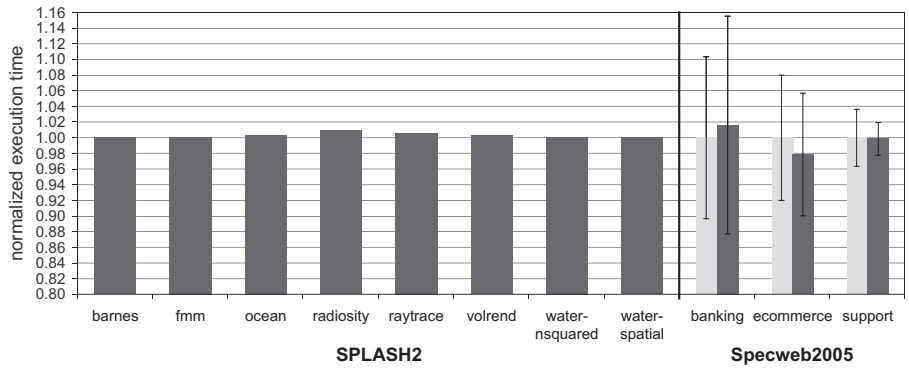

Fig. 9. Normalized execution time of the one-bit improved ID filter compared to the system without filter.

Section 6.1) for the system with and without the one-bit improved ID filter. Because of this, Specweb2005's benchmarks have two bars: the first one represents the mean and standard deviation for the system without filtering, and the second one corresponds to the mean and standard deviation in the system using the one-bit improved ID filter.

For SPLASH2, Fig. 9 shows that, on average, the system is 0.3% slower when using the one-bit improved ID filter. `Radiosity` is the benchmark with the worst performance. It has a performance loss of 0.9%. In this benchmark, the compiler allocated frequently accessed constants in the code region. Thus, some shared cache blocks are accessed simultaneously by loads and instruction fetches. In the original coherence protocol, the subblocks accessed only by instruction fetches or only by loads are cached in the local caches. In contrast, when forcing the instruction/data exclusivity at a 64B granularity, not all those subblocks are locally cached. Thus, some loads or instruction fetches need to access the shared cache instead of only the local caches.

Fig. 9 shows that in Specweb2005 we can differentiate two groups: for `banking` the mean execution time shows an increase of 1.6%; for `ecommerce` and `support`, the mean execution time shows a decrease of 2.1% and 0.1%, respectively. In both groups the confidence interval shows that the execution time is not statistically different.

For `banking`, the variability is high because we can not simulate enough web transactions due to simulation time restrictions. As the number of web transactions is low, the cold-start and end-effect may influence the results. The first transaction to complete within the interval would have started before the interval began. Similarly, when the last transaction completes, the next ones would have already started. In order to reduce the confidence intervals, we executed more simulation runs of Banking than of the rest of Specweb2005 workloads.

6.4. Power reduction

CACTI 6.5 [19] is used to estimate dynamic energy and leakage power for the cache tag array and the proposed filters. We modified CACTI to model CAM structures, so that dynamic directory energy and leakage power can be estimated. All structures were modeled using a 65 nm technology with a target frequency of 1.2 GHz.

The average dynamic power consumption is computed based on activity statistics of the shared cache, the filters, and the data and instruction directories along the execution of the benchmarks. The average dynamic power consumption of the directory is 1.5 times the average dynamic power consumed by the tag array of the shared cache. However, the leakage power of the tag array is 2.2 times the directories leakage since the tag array is bigger than the directory structure.

Fig. 10 shows the percentage of power reduction in the directory using the different proposed filters. The directory power consumption includes the dynamic power and the leakage power in both data and instruction directories. There are three columns for each benchmark. Each column corresponds to the directory power reduction when using one of the proposed filters: the two-bit ID filter, the one-bit improved ID filter, and the Bloom filter (from left to right).

For SPLASH2, on average, the power reduction is quite similar when using any of the proposed filters. The two-bit and the one-bit improved ID filters, on average, reduce the power consumption by 28%. The Bloom filter reduces the directory power by 27%. In contrast, for Specweb2005, there is an important difference between the reduction achieved by the ID filters and the Bloom filter. Fig. 8 shows that the Bloom filter identifies fewer useless directory lookups than the other proposed filters. Thus, the directory power is only reduced by 9% when using the Bloom filter. On the other hand, any of the ID filters, on average, reduce the directory power by 19% for Specweb2005.

There are important differences on average power reduction between SPLASH2 (28%) and Specweb2005 (19%), though Fig. 8 shows that the ID filters are as effective for SPLASH2 as for Specweb2005. For `banking` the average power reduction is similar to the reduction observed for SPLASH2, but `ecommerce` and `support` experience a lower reduction. The differences are due to simulating Specweb2005 in single-threaded processors. `Ecommerce` and `support` have a high shared cache miss rate. This high miss rate, together with the existence of just one thread per core, give rise to frequent core stalls in which no request is sent to the shared cache. As long as the core is stalled, no directory lookups are performed. Thus, during core stalls, filters are not filtering out directory lookups to reduce power consumption. However, the filter consumes leakage power. The dynamic power reduction in the directory is smaller due to less accesses, but the increase in the leakage power due to the filter remains the same. To prove this argument we simulate SPLASH2 suite in a system with single-threaded cores and we observe a similar reduction in saved power.

On average, the one-bit improved ID filter gets a slightly bigger reduction than the rest of the proposed filters. The drawback of this filter is that it comes with a small performance loss (Fig. 9).

6.5. Other cache sizes and new generation technologies

The size of the ID filters is directly proportional to the number of blocks in the shared cache. An increase in the shared cache size, keeping the same block size, increases the number of blocks and so the number of entries of the ID filters. Therefore, the leakage and dynamic power consumption of the filter also increase.

The Bloom filter size is determined by three parameters: the number of entries, the number of hash functions, and the number of bits of each entry (counter bits). The counter bits should be enough to count all blocks allocated in the directory the Bloom filter is associated to. Thus, an increase in the number of blocks allocated in the directory will increase the counter bits. The

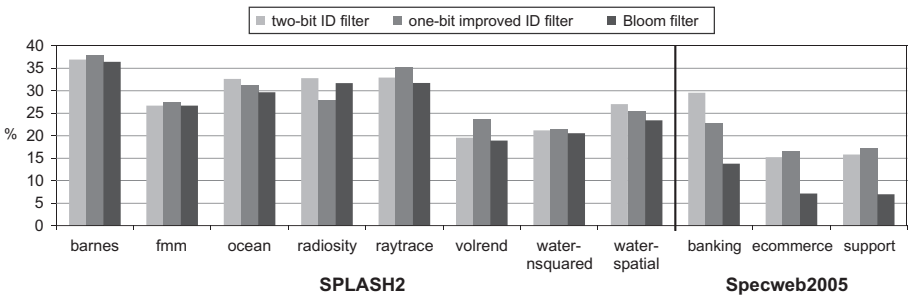

Fig. 10. Percentage of power reduction in the directory.

number of entries and the number of hash functions should be big enough to keep the number of false positives low. An increase in the directory size increases the number of blocks allocated in the Bloom filter. As a result, the number of false positives in a specific Bloom filter increases. It is then necessary to check that the accuracy loss in the Bloom filter does not affect the results.

Fig. 11 shows the average percentage of power reduction for SPLASH2 and Specweb2005 for different local and shared cache sizes for each proposed filter. Both benchmark suites have three groups of columns: the two-bit ID filter, the one-bit improved ID filter, and a Bloom filter with 128 entries and 2 hash functions (from left to right). There are 4 columns in each group. The first one shows the average numbers presented in Fig. 10 that are for the memory hierarchy parameters defined in Table 1. For the next columns, the size of the local and shared caches is increased, but the rest of their parameters remain the same as before. In the second column, only the shared cache size is doubled, in the third column only the local caches are doubled and in the fourth one both local and shared cache sizes are doubled.

The number of shared cache evictions is reduced when the shared cache size is doubled. Thus, the energy consumed by the directory is reduced, so the percentage of power reduction is also reduced. Our workloads are barely affected by this effect since the number of evictions is small for both shared cache sizes.

When the shared cache size is increased, the power consumption of the ID filters increases. We expect the percentage of power reduction to decrease. Fig. 11 shows that the percentage of power reduction is only reduced for the two-bit ID filter. The one-bit improved ID filter barely increases the leakage and power consumption of the tag array, so an increase in the shared cache size does not affect the percentage of power reduction in the directory.

Fig. 11 also shows that for the Bloom filter the percentage of power reduction is not affected when the shared cache is doubled as it was expected.

When the local cache size is doubled, Fig. 11 shows that all filters decrease the percentage of power reduction. When the local caches size is doubled, both the directory leakage and dynamic power increase. However, the directory leakage is almost multiplied by a factor of 3 while the dynamic power is only multiplied by a factor of 2. Such an important increase in the leakage power affects the percentage of power reduction in the directory for all filters.

Finally, we analyze how the percentage of power reduction is affected for new generation technologies. Fig. 12 compares the percentage of power reduction when using 65 nm and 22 nm technologies. We use the memory hierarchy parameters described in Table 1 and a Bloom filter with 128 entries and 2 hash functions. Fig. 12 shows average numbers for SPLASH2 and Specweb2005. There are three groups of columns for each benchmark suite: the two-bit ID filter, the one-bit improved ID filter, and the Bloom filter (from left to right). There are 2 columns in each group. The first one shows numbers for 65 nm technology with a target frequency of 1.2 GHz and the second column shows numbers for 22 nm technology with a target frequency of 2.75 GHz.

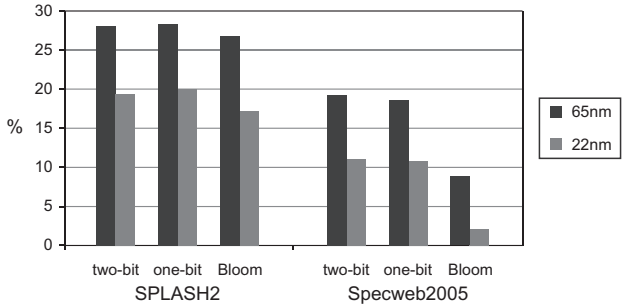

Fig. 12. Percentage of power reduction in the directory modeling all the structures with a 65 nm technology or a 22 nm technology.

Fig. 12 shows that the power reduction when using the different filters is smaller for a 22 nm technology than for a 65 nm technology. In SPLASH2, the power reduction attained is still quite interesting for all the proposed filters. The Bloom filter shows the worse result reducing the power by 17%, and the one-bit improved ID filter gets the best result reducing by 20% the power. In Specweb2005, the power is reduced by 11% when using the two-bit and one-bit improved ID filters. However, the Bloom filter does not get such good results. It only reduces the power by 2%.

7. Related work

During the last decade several techniques to filter out coherence actions in snoopy-based protocols have been published. The proposed mechanisms try to reduce either local cache lookups performed by coherence requests or directly the broadcast messages. In order to reduce the coherence actions a filtering structure is necessary. This structure is either placed together with the local caches or distributed across the on-chip network.

When the filter is placed together with the local cache in snoopy-based protocols in bus-based systems, we can distinguish several ways to reduce the power consumed by coherence actions.

Several proposals try to filter snoop-induced lookups. JETTY [6] adds small structures to SMPs that are accessed before doing the tag cache lookup. Ekman et al. [20] evaluate JETTY on CMPs and conclude that, as the local caches are smaller than in SMPs, JETTY is not an interesting mechanism for CMP systems because the energy consumption of the filters and the local caches are similar. Salapura et al. [21] propose a structure that keeps a superset of cached blocks. The Page Sharing Table (PST), proposed by Ekman et al. [22], uses vectors that identify sharing at the page level with precise information.

There is a group of proposals trying not only to filter snoop-induced lookups but to reduce broadcast messages. RegionScout

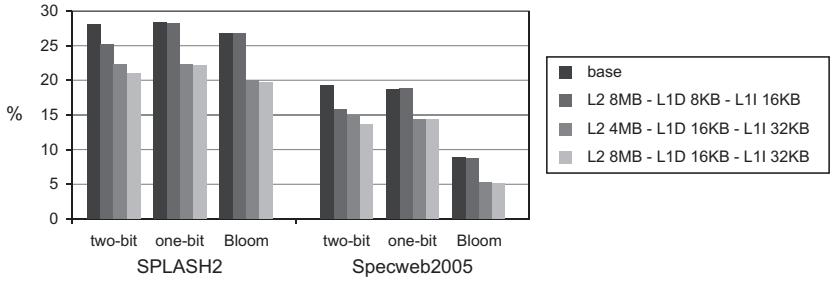

Fig. 11. Percentage of power reduction in the directory with different local and shared cache sizes.

[23] implements several structures per node in a similar way to JETTY [6], but these structures keep global system information about regions, which are continuous sections of memory. Cantin et al. [24] present an idea similar to RegionScout, but the information kept in the structures is precise and the structures are bigger.

Focusing on logical ring interconnections, Strauss et al. [25] propose using an adaptive filter in each node to skip the snoop-induced lookup when possible and to decide if the lookup should be performed in parallel to sending the request to the next node (to reduce snoop latency) or in sequence (to reduce the number of messages).

Compiler time knowledge can also be used to reduce coherence actions. Information about the behavior of a program helps determining whether a region of memory is shared or private, thus limiting snoop-induced lookups to shared blocks [26,27].

There are other proposals that distribute the filter over the on-chip network for snoopy-based. Agarwal et al. [28] propose adding a region tracker structure in each output port of the routers. This structure indicates which regions are not allocated in the local caches of the processors reached from a specific port, so useless broadcast messages are not sent.

Lately, techniques to filter coherence actions in directory-based protocols have also appeared. Lotfi-Kamran et al. [29] present TurboTag, which is a filter that reduces the number of directory lookups performed in a system with write-back local caches. Their filter, placed together with the directory, is based on a Bloom filter. There are also other proposals that, like in snoopy-based protocols, distribute the filter over the on-chip network. Jerger [30], in a coarse-grain like directory-based protocol, adds Counting Bloom Filters [13] to each output port of the routers in order to not broadcast useless invalidation messages addressed to the local caches reached from a specific port.

Our proposed filter tries to reduce the number of directory lookups in a directory-based protocol. Unlike previous proposals for directory-based protocols, our filtering mechanism is designed for a system which, like Niagara 2, has write-through local caches and the directory is implemented as a duplicate tag directory. TurboTag [29] proposes the closest filtering mechanism since it filters lookups to a duplicate tag directory and uses a filter based on a Bloom filter. However, TurboTag is designed for systems with write-back local caches. In our CMP model, the number of stores that access the shared cache is bigger than in a system with write-back local caches and the directory lookups performed by these extra stores are not useless, that is, there is a copy of the target block in a local cache (and therefore, in the directory). Consequently, as our experiments have shown, ID filters are more efficient than Bloom based filters for systems with write-through local caches.

8. Conclusions

An important fraction of directory lookups are useless because there are no copies of the target block in any local cache in the system. We could decide not to perform these directory lookups and program execution would remain correct. These useless directory lookups waste energy, but in a directory coherence mechanism there is no way to avoid them. We propose to use a filter before accessing the directory which is able to identify in advance whether a lookup is useless or not.

We propose two basic filter implementations. In the first implementation, we exploit the inclusion property of the shared cache to label each block with the stream it belongs to (data or instruction). In the second one, we keep the information of all blocks belonging to a stream together using a separate Bloom filter for each directory (data and instruction).

We propose three different filters based on the two implementations described: the two-bit ID filter, the one-bit improved ID filter, and the Bloom based filter. Using any of these filters, on average, more than 60% of the directory lookups are avoided for SPLASH2. For Specweb2005, the two-bit and one-bit improved ID filters reduced the directory lookups by more than 60%, but the Bloom filter only reduces them by 45%. The two-bit and one-bit ID filters achieve 28% and 19% reduction in power consumption for SPLASH2 and Specweb2005, respectively. When using the Bloom filter the power consumption is reduced by 27% and 9% for SPLASH2 and Specweb2005, respectively.

The results shown in this paper lead to the conclusion that ID filters perform better than Bloom based filters. From the power consumption perspective, although both filters have a similar energy consumption by construction, ID filters are able to avoid more directory lookups than Bloom based filters. A good advantage of Bloom based filters over ID filters is that their size do not grow with the size of the shared cache. However, when analyzing power consumption for large shared cache sizes, we see that attained power consumption of the one-bit improved ID filter is independent of the shared cache size. As a result, the one-bit improved ID filter is the best solution proposed, outperforming the other analyzed implementations both in terms of performance and energy consumption.

Acknowledgments

This work was supported in part by Grants TIN2010-21291-C02-01 and TIN2007-60625 (Spanish Government), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government and European ERDF), and HiPEAC-2 NoE (European FP7/ICT 217068).

References

[1] H.Q. Le, W.J. Starke, J.S. Fields, F.P. O'Connell, D.Q. Nguyen, B.J. Ronchetti, W.M. Sauer, E.M. Schwarz, M.T. Vaden, IBM POWER6 microarchitecture, IBM J. Res. Dev. 51 (6) (2007) 639–662.
[2] AMD, "AMD Multi-Core Technology", <http://multicore.amd.com>.
[3] Fujitsu, "Fujitsu SPARC64 VII Processor," June 2008.
[4] Intel, "Leading Virtualization Performance and Energy Efficiency in a Multiprocessor Server".
[5] T. Johnson, U. Nawathe, An 8-core, 64-thread, 64-bit Power Efficient SPARC SOC (niagara2), in: ISPD '07, 2007, pp. 2–2.
[6] A. Moshovos, G. Memik, B. Falsafi, A. Choudhary, JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers, in: HPCA-7, 2001, pp. 85–96.
[7] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz, An Evaluation of Directory Schemes for Cache Coherence, in: ISCA-15, May–2 Jun 1988, pp. 280–289.
[8] L. Censier, P. Feautrier, A new solution to coherence problems in multicache systems, Comput. IEEE Trans. C-27 (12) (1978) 1112–1118.
[9] C.K. Tang, Cache System Design in the Tightly Coupled Multiprocessor System, in: AFIPS '76, 1976, pp. 749–753.
[10] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (1970) 422–426.
[11] OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification, vol. 1, Sun Microsystems, Inc., May 2008.
[12] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, B. Verghese, Piranha: a Scalable Architecture Based on Single-Chip Multiprocessing, in: ISCA-27, 2000, pp. 282–293.
[13] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol 8 (2000) 281–293.
[14] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: a full system simulation platform, Computer 35 (2) (2002) 50–58.
[15] J.P. Singh, A. Gupta, M. Ohara, E. Torrie, S.C. Woo, The SPLASH-2 Programs: Characterization and Methodological Considerations, ISCA-22, 1995, p. 24.
[16] SPEC, Specweb2005 Release 1.10 Benchmark Design Document, Technical Whitepaper, 2006.
[17] M. Monchiero, J.H. Ahn, A. Falcón, D. Ortega, P. Faraboschi, How to simulate 1000 cores, SIGARCH Comput. Archit. News 37 (2) (2009) 10–19.
[18] A.R. Alameldeen, D.A. Wood, Variability in architectural simulations of multi-threaded workloads, in: HPCA-9, 2003, p. 7.
[19] N. Muralimanohar, R. Balasubramonian, CACTI 6.0: A Tool to Model Large Caches, Technical report, HP Laboratories Palo Alto, 2009.

[20] M. Ekman, F. Dahlgren, P. Stenström, Evaluation of snoop-energy reduction techniques for chip-multiprocessors, in: Workshop on Duplicating, Deconstructing and Debunking, 2002. in Conjunction with ISCA, May 2002.
[21] V. Salapura, M. Blumrich, A. Gara, Improving the accuracy of snoop filtering using stream registers, in: MEDEA '07, 2007, pp. 25–32.
[22] M. Ekman, P. Stenström, F. Dahlgren, TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors, in ISLPED'02, 2002, pp. 243–246.
[23] A. Moshovos, RegionScout: exploiting coarse grain sharing in snoop-based coherence, in: ISCA-32, June 2005, pp. 234–245.
[24] J. Cantin, M. Lipasti, J. Smith, Improving multiprocessor performance with coarse-grain coherence tracking, in: ISCA-32, June 2005, pp. 246–257.
[25] K. Strauss, X. Shen, J. Torrellas, Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors 34 (2) (2006) 327–338.
[26] A. Dash, P. Petrov, Energy-efficient cache coherence for embedded multiprocessor systems through application-driven snoop filtering, in: DSD '06, 2006, pp. 79–82.
[27] C.S. Ballapuram, A. Sharif, H.-H.S. Lee, Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors, in: ASPLOS XIII, 2008, pp. 60–69.
[28] N. Agarwal, L.-S. Peh, N. Jha, In-network coherence filtering: snoopy coherence without broadcasts, in: MICRO-42, 2009, pp. 232 –243.
[29] P. Lotfi-Kamran, M. Ferdman, D. Crisan, B. Falsafi, TurboTag: lookup filtering to reduce coherence directory power, in: ISLPED '10, 2010, pp. 377–382.
[30] N. Jerger, SigNet: network-on-chip filtering for coarse vector directories, in: DATE'10, 2010, pp. 1378–1383.

Ana Bosque is a PhD candidate in the IIS Department at the University of Zaragoza, although she is based on Barcelona, working as research assistant in the Computer Architecture Department at the Universitat Politècnica de Catalunya (UPC). She has a MS in Computer Engineering from the University of Zaragoza (2002). Her research interests include memory hierarchy and coherence protocols for chip multiprocessors.

Victor Viñals received the MS degree in Telecommunication, and the PhD degree in Computer Science from the Universitat Politècnica de Catalunya (UPC) in 1982 and 1987, respectively. He was associate professor in the Facultat d'Informática de Barcelona (UPC) in the 1983–1988 period. Currently, he is full professor in the IIS Department at the University of Zaragoza (Spain). His research interests include processor microarchitecture, memory hierarchy and parallel computer architecture. He is member of the ACM and the IEEE Computer Society. He also belongs to the Juslibol Midday Runners Team and to the Computer Architecture Group of the University of Zaragoza.

Pablo Ibáñez is an associate professor in the IIS Department at the University of Zaragoza. His research interests include memory hierarchy, processor microarchitecture, and parallel computer architecture. He has an MS in computer science from Universitat Politècnica de Catalunya and a PhD in computer engineering from the University of Zaragoza. Ibáñez is a member of the IEEE, the ACM, and the I3A.

José M. Llabería received the MS degree in telecommunication, and the MS and the PhD degrees in computer science from the Universitat Politècnica de Catalunya (UPC) in 1980, 1982, and 1983, respectively. He is a full professor in the Computer Architecture Department at UPC (Barcelona, Spain). His research interests include processor microarchitecture, memory hierarchy, parallel computer architecture, vector processors, and compiler technology for these processors.