2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools

Filtering Directory Lookups in CMPs

Ana Bosque * Víctor Viñals ‡ Pablo Ibáñez ‡ José M. Llabería *

* *Computer Architecture Department*
Uniersitat Politècnica de Catalunya
Barcelona, Spain
{*abosque, llaberia*}*@ac.upc.edu*

‡ *Department of Computer Science*
and Systems Engineering
University of Zaragoza
Zaragoza, Spain
{*victor, imarin*}*@unizar.es*

***Abstract*—Coherence protocols consume an important fraction of power to determine which coherence action should take place. In this paper we focus on CMPs with a shared cache and a directory-based coherence protocol implemented as a duplicate of local caches tags. We observe that a big fraction of directory lookups produce a miss since the block looked up is not cached in any local cache. We propose to add a filter before the directory lookup in order to reduce the number of lookups to this structure. The filter identifies whether the current block was last accessed as a data or as an instruction. With this information, looking up the whole directory can be avoided for most accesses.**

We evaluate the filter in a CMP with 8 in-order processors with 4 threads each and a memory hierarchy with a shared L2 cache. We show that a filter with a size of 3% of the tag array of the shared cache can avoid more than 70% of all comparisons performed by directory lookups with a performance loss of just 0.2% for SPLASH2 and 1.5% for Specweb2005. On average, the number of 15-bit comparisons avoided per cycle is 54 out of 77 for SPLASH2 and 29 out of 41 for Specweb2005. In both cases, the filter requires less than one read of 1 bit per cycle.

I. Introduction

During the past decade single-core processors have become increasingly more complex reaching a point of diminishing returns on power/performance. This fact, along with the always increasing density in the chips, have encouraged the development of multi-core chips. Nowadays, most computer manufacturers offer multi-core chips such as the IBM Power6 [1] with two cores, the AMD Phenom II [2] with four cores, the Fujitsu SPARC64 VII [3] with four cores, the Intel Xeon 7400 series [4] with six cores, and the SUN Niagara 2 [5] with eight cores. In all of them there is at least a local cache level per node that is kept coherent by means of a coherence protocol.

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 should take place. In the snoopy-based protocols the state of each block is stored in the local caches, that is, the information about the state of the cached data is distributed. As a result, all transactions should be sent to all the local caches in the system.

Both kinds of protocols consume an important fraction of shared cache energy to determine the action to take. In snoopy-based protocols the energy is spent on broadcasting coherence messages and making tag-cache lookups [6]. Directory-based protocols reduce energy consumption compared to snoopy-based protocols because it is known which caches have a copy of a block [7]. A directory implemented as a copy of the local cache tags requires a small amount of area, but the lookups in this structure are highly associative. For example, Niagara 2 follows the latter scheme and a directory lookup can perform up to 256 15-bit comparisons.

Along the last decade there have been several proposals to reduce the power consumed by snoopy-based protocols [6], [8], [9], [10], [11], [12]. Most of these proposals are based on the fact that most broadcasts and tag-cache lookups are not necessary, so it is possible to use filters that discard actions that will be useless.

In directory-based protocols, we have observed that an important fraction of the accesses performed to the coherence directory "misses", that is, the searched block is not cached in any local cache in the system. In these situations, the energy consumed by the directory access is wasted. We propose to use simple filters to reduce unnecessary and energy consuming accesses to the directory in CMPs like Niagara 2 [5] where the local cache level is split into an instruction cache and a data cache and the shared cache is inclusive.

Although instruction and data streams generally access different memory regions, it is impossible to assure whether 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, in every search in a directory implemented as a copy of the local tags 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 a CMP like Niagara 2 with an inclusive shared cache, we can exploit the inclusion property and the knowledge

978-0-7695-4171-6/10 $26.00 © 2010 IEEE
DOI 10.1109/DSD.2010.85

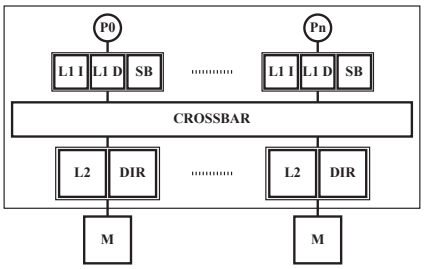

Figure 1: CMP model having a first-level local cache per core (instruction cache, data cache and store buffer) and a second-level shared cache divided in several banks.

Table I: Memory hierarchy parameters

L1 D size	8KB	L2 size	4MB
L1 D associativity	4-way	L2 number of banks	8
L1 D block	16B	L2 associativity	16-way
L1 I size	16KB	L2 block	64B
L1 I associativity	8-way	L2 latency	7 cycles
L1 I block	32B	L2 MSHR	8
Crossbar arbitration	3 cycles	SB	8 entries per thread
Crossbar latency	3 cycles		
Physical address	40 bits	Memory latency	117 cycles

of the type of the shared cache access (instruction fetch, data, store) to implement a filter that reduces the number of associative lookups. Thus, the filters proposed in this paper identify the stream (data or instruction) that the accessed block belongs to. Based on this information, the number of directory lookups can be greatly reduced.

The rest of this paper is organized as follows. In Section II, we describe in detail the memory hierarchy of the chosen CMP model. We motivate our idea in Section III. In Section IV we explain how the proposed filters work. Section V shows our experimental results and Section VI discusses related work. Finally, Section VII contains the conclusions.

II. Chip multiprocessor model

Figure 1 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 per core, and is composed of an instruction cache (L1 I) and a write-through data cache (L1 D). Each core has also a store buffer (SB) with several entries per thread that contains all outstanding stores. The second-level cache (L2), which is inclusive, is shared among all the cores. It is divided in different banks interleaved by second-level cache blocks. A crossbar communicates the two cache levels. A write-invalidate directory-based protocol is used to maintain cache coherence. The directory is distributed among the second-level cache banks, keeping close to every bank the information about the blocks associated with it. Table I collects the specific parameters we chose for the memory hierarchy. All of them are based on Niagara 2.

We also assume a directory similar to that of Niagara 2 [13], which consists of a copy of the local cache tags. In each bank, the directory 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. As there is inclusion between the local caches and the shared cache and the directory is accessed after the shared cache tag array, the area of the directory is reduced by keeping pointers to the shared cache instead of the local cache tags, as only the set index and the way in the shared cache has to be stored.

The directory is split into instruction and data directories, replicating the organization of the local caches. A directory gives the way or ways of the local caches where the copies of the subblocks are located. Thus, an invalidation message consists of the set index in the local cache and the way in the set. A cache block in a local cache is invalidated by unsetting the valid bit of the way where it is located. Stores update local caches when the ack message is received. The message includes the way where the copy of the block is located in order to eliminate the lookup in the local cache.

Like in Niagara 2 [13], 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 to ensure the instruction/data exclusivity. It is important to notice that, as the block size of the local caches is smaller than the block size of the shared cache, there can be copies of different parts of a shared cache block in local caches of different types.

A. Directory Organization in a Bank

The duplicate instruction and data directories have a similar structure (see Figure 2), but they are accessed in a different way depending on the kind of memory operation.

The 8 banks of the shared cache are interleaved by 64B block (bits 8:6). The local data cache has 128 sets, so four groups of sets (bits 10:9) in a local data cache are mapped to a bank of the shared cache. As the block size of the shared cache is 4 times the block size of a local cache, each of the four groups consists of four contiguous sets (bits 5:4). Thus, the data directory of a bank of the shared cache has 512 blocks (4 groups x 4 sets per group x 4-way x 8 cores). The directory is organized as a set of 16 panels (using SUN's terminology [13]). A panel is a group of blocks with the same set index in the local caches. As any cached block can be allocated in 8 processors (cpu_id) and the local data cache is 4-way associative, a panel is a CAM of 32 entries of 15 bits each. The panels are arranged in 4 rows (contiguous sets) and 4 columns (group of sets).

In a similar way, the instruction directory of a shared cache bank tracks 512 blocks (4 groups x 2 sets per group x 8-way x 8 cores). It is also organized in panels of the same

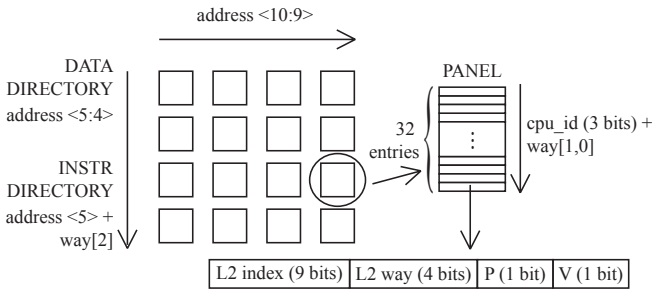

Figure 2: Data or instruction directory structure and how they are accessed.

size as the data directory. Differences are due to larger cache block size and higher associativity in the local instruction caches.

III. Motivation

Any access and any eviction done in 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. Below we describe the memory operations and the actions performed in the directory for each one:

- *Load missing in the local cache ("load-miss" from now on)*: 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 (L1 D block size) in all the local instruction caches. The address of this 16B block determines the two single instruction directory panels that can contain a copy of it. These two panels are looked up.
- *Instruction fetch missing in the local cache ("ifetch-miss" from now on)*: 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 (L1 I block size) are invalidated in the local data caches. Thus, two subblocks of 16B are invalidated. The address of these subblocks determine two data directory panels.
- *Store*: As the local data cache is write-through, every executed store access 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 data directory panel and two instruction directory panels.
- *Eviction*: As inclusion is enforced in the system, the shared cache victim block has to be removed from the local caches, so 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 (L2 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. These blocks determine four panels in each directory.

Table II: Actions performed in the directories for every memory operation that access the shared cache. For each lookup action, the number of panels looked up 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 (2)
ifetch-miss	lookup (2)	update
store	lookup (1)	lookup (2)
eviction	lookup (4)	lookup (4)

Table II summarizes the actions performed for every memory operation and the number of panels accessed in the lookup actions. The shaded 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. Lookup actions are only useful if they hit, that is, if the block looked up 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.

Figure 3 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 V for a description of the used workloads and the simulated system. Due to space restrictions, we represent only the average of the three workloads of Specweb2005. As not all lookups represent the same amount of comparisons, we represent the *number of directory panels looked up*, instead of the number of directory lookups.

Figure 3 has three columns for each benchmark. The first one shows the memory operations categorized as load-misses, ifetch-misses, stores and evictions. The second column corresponds to the data and instruction directory panel lookups generated by the memory operations of the first column. The last column represents the number of "useful" data and instruction directory panels looked up.

The number of directory panels looked up, which requires 32 comparisons per panel, is, on average, almost three times the number of memory operations, but only 22% of them for SPLASH2 and 15% for Specweb2005 are useful. More than 70% of the memory operations are stores for both SPLASH2 and Specweb2005. Thus, the number of panel lookups necessary for maintaining instruction/data exclusivity (shaded cells in Table II) represent a small fraction (17% on average) of the total number of panel lookups.

In SPLASH2, 28% of the directory panel lookups are performed in the data directory and 76% of them are useful.

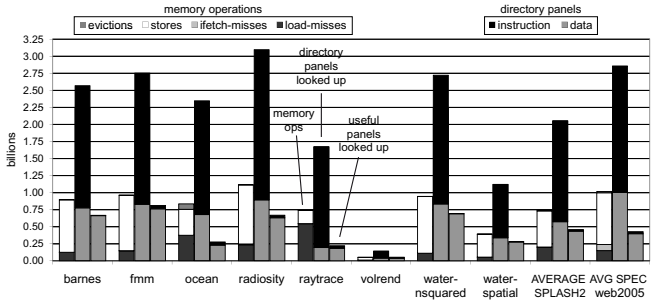

Figure 3: The first column for each benchmark represents the billions of memory operations that access the shared cache categorized as load-misses, ifetch-misses, stores and evictions. The second column collects the billions of panels looked up in each directory. The third column represents the billions of "useful" directory panels looked up in each directory.

The rest of directory panel lookups (72%) are performed in the instruction directory and only 2% of them are useful. For Specweb2005, the instruction directory panels looked up represent a smaller fraction than in SPLASH2 (65%) and the fraction of useful ones is the same as in SPLASH2 (2%). The rest of directory panel lookups are performed in the data directory (35%) and the fraction of useful ones is smaller than in SPLASH2 (40%). This decrease in the number of useful data directory panel lookups is a result of the larger number of ifetch-misses that access the shared cache in Specweb2005. These memory operations perform data directory panel lookups that, in general, are not useful. In SPLASH2, on the contrary, most of the data directory panel lookups are performed by stores and, in general, they are useful.

Results from Figure 3 clearly indicate that if we know in advance whether a directory panel lookup is going to be useful or not, the number of comparisons (and hence the energy consumption) can be greatly reduced.

IV. Filtering by the type of the block

In general, in the execution of any program, the set of memory addresses of the instructions executed and the set of memory addresses of data accessed are disjunctive.

In our CMP model the local caches are split into data and instruction caches. 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 Section III we saw that both stores and evictions need to look up both directories because we can not assure that instruction and data are always located in different memory regions. Two examples of this situation are self-modifying code and locating constants in the code segment. This last case takes place when a compiler locates program constants along with the instructions using them. In this situation, 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.

We propose to implement a filter that is able to know if a block in the inclusive shared cache has been accessed only for instruction-fetch or only as data. With this information, lookup actions would either be avoided or be performed in one and only one of the directories (data or instruction), thus reducing the energy consumed. We call this filter "instruction-data filter" (ID filter).

A. ID Filter Operation

The ID filter is implemented as metadata associated with each block in the inclusive shared cache, similar to the state bits of the block. For every memory operation, the filter is read in parallel with the L2 tag array. Then, if necessary, the directory is accessed in parallel with the L2 data array. Note that if the directory is implemented using pointers to the shared cache instead of the local cache tags themselves, it is already necessary to access the tag array before accessing the directory.

By using the ID filter, the update actions performed in the directory by load-misses and ifetch-misses (Table II) can not be avoided. However, lookup actions can be greatly reduced.

B. 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 will be marked as *data blocks*; blocks that have been accessed only by instruction fetches will be marked as *instruction* blocks; and blocks accessed by instruction fetches and stores or loads will be 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 they are updated only when the type of the block changes. A data block changes its type when it is accessed by an ifetch-miss and an instruction block changes its type when it is accessed by a load-miss or a store. In both cases the type of the block changes to mixed.

Table III collects the actions performed in the directory depending on the memory operation and the type of the accessed block. Comparing Table II and Table III, 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 marked by the ID filter as belonging to the instruction or data stream, it is only necessary to look up in one directory. For a data block, all its copies must be in the local data caches, so only the data directory is looked up.

Table III: Actions performed in the directories when using the two-bit ID filter. For each lookup action, the number of panels accessed is enclosed

memory operation	block type	data directory	instruction directory
load-miss	data	update	—
	instr	update	lookup (2)
	mixed	update	lookup (2)
ifetch-miss	data	lookup (2)	update
	instr	—	update
	mixed	lookup (2)	update
store	data	lookup (1)	—
	instr	—	lookup (2)
	mixed	lookup (1)	lookup (2)
eviction	data	lookup (4)	—
	instr	—	lookup (4)
	mixed	lookup (4)	lookup (4)

For an instruction block, all its copies must be in the local instruction caches and only the instruction directory should be looked up.

In the two-bit ID filter, a block classified as mixed can not change its type as long as it remains in the shared cache. As an example, for an instruction block that is accessed as data once, even though hundreds of instruction fetches of it take place, it will not be considered an instruction block anymore. This situation could be harmful for the filter performance if the block is highly accessed.

C. A smaller filter: the one-bit 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 contains 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 (L2 block size). So, each block in the shared cache is forced to be classified either as a data block or as an 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 needs to invalidate all its copies in the local data caches. Likewise, if it changes from instruction to data, it is needed to invalidate its copies in the local instruction caches. In order to carry out the invalidations, a directory lookup is needed. We expect the energy consumed by these new directory lookups to be small enough to not spoil the overall benefit got by the instruction/data exclusivity.

Table IV shows the actions performed in the directory for each memory operation when instruction/data exclusivity is forced. Comparing Table III and Table IV, we see that,

Table IV: Actions performed in the directories after looking up in the filter if instruction/data exclusivity for 64B blocks is forced.

memory operation	block type	data directory	instruction directory
load-miss	data	update	—
	instr	update	lookup (4)
ifetch-miss	data	lookup (4)	update
	instr	—	update
store	data	lookup (1)	—
	instr	—	lookup (4)
eviction	data	lookup (4)	—
	instr	—	lookup (4)

in general, forcing data/exclusivity for 64B blocks causes a bigger number of panel lookups when the memory operation changes the type of the block: when a ifetch-miss access a data block or when a load-miss or store access an instruction-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.

In this implementation, as in the previous one, the filter information 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.

V. Evaluation

A. Methodology

We use a Simics-based simulator in which we have modeled, using the tools provided by Simics, a memory hierarchy with the parameters described in Table I. Simics[14] is a full-system multiprocessor simulator capable of running unmodified commercial OSs and applications. We configured Simics to model a SPARC V9 target system with a Total Store Order (TSO) consistency memory model running Solaris 9. We simulated a system with 8 in-order, blocking, 1.2GHz 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: 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. [16]. For water-nsquared and water-spatial we were only able to scale the datasets to 2k and 4k particles respectively to bound the simulation time. We execute the whole parallel section of each benchmark. Table V shows the applications used, the corresponding datasets and the billions of executed instructions.

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

benchmark	dataset	instr (10^9)	cycles (10^9)
barnes	64K particles	4.97	0.62
fmm	64K 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 VI: Specweb2005 workloads, the corresponding simultaneous sessions, the number of web transactions and billions of instructions executed.

workload	simultaneous sessions	web transactions	instr (10^9)
Banking	200	100	15.52
Ecommerce	1000	1200	8.07
Support	1400	2200	8.07

For the three workloads of Specweb2005, we use the web server Apache 2.0.63. After the initialization of the workload, we warm the caches during 0.75 billions of cycles and then simulate during 2.25 billions of cycles. The thinking time is always zero. Table VI shows the number of transactions and billions of instructions executed for each workload.

B. Coverage

In Figure 3, we showed that a big amount of lookups are useless. Now, we analyze whether the ID filters proposed are able to identify the useless lookups in advance or not.

Figure 4 shows the number of directory panel lookups in the system without filter and when using the ID filters. For each benchmark there are six columns. The first three columns correspond to the instruction directory and the next three correspond to the data directory. In both groups, the first column shows the total number of directory panel lookups in the system without filter, the second column represents the number of directory panel lookups when using the two-bit ID filter and the third column shows the number of directory panel lookups when using the one-bit ID filter.

Figure 4 shows that for Specweb2005, the two-bit ID filter identifies 98% of the instruction directory panel lookups and 39% of the data directory panel lookups as useless. The one-bit ID filter identifies 99% of the instruction directory panel lookups, but only 16% of the data directory panel lookups.

For SPLASH2, the instruction directory panel lookups identified as useless are 98% of the total for both ID filters. The two-bit ID filter identifies 4% of the data directory panel lookups as useless, but the one-bit ID filter requires an additional 2% data directory panel lookups to work properly.

Summing up, the directory panel lookups (i.e., instructions + data) identified as useless by the two-bit ID filter are 72% of the total panels looked up for SPLASH2 and 77% for Specweb2005. The one-bit ID filter identifies as useless 70% of the total panels looked up for both benchmark suites. For SPLASH2, the two-bit ID filter identifies 92% of the useless panel lookups and the one-bit ID filter 90%. For Specweb2005, the two-bit ID filter identifies 91% of the useless panel lookups and the one-bit ID filter 82%.

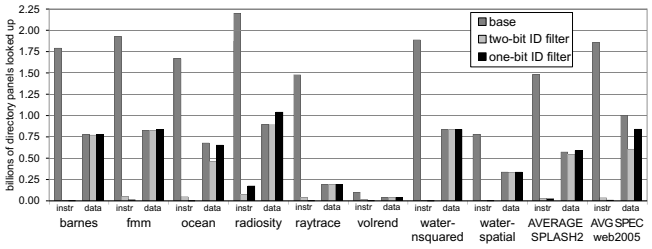

Figure 4: Billions of directory panel lookups in the system without filter and when using the two-bit and one-bit ID filters. The first three columns correspond to the instruction directory and the next three correspond to the data directory.

Both ID filters are able to filter a small fraction of data directory panel lookups. The reason is that, in general, the data directory panel lookups are useful, so the filters should not avoid them. Most of the data directory panel lookups are performed by stores. Stores represent an important fraction of the memory operations in the shared cache because local data caches in our memory hierarchy are write-through. Data directory panel lookups generated by stores are, in general, useful so they should not be filtered out. However, instruction directory panel lookups generated by stores, which are 2x the number of data directory panel lookups (recall Table II) are not useful and can be safely filtered out. In Specweb2005, as an important fraction of the data directory panel lookups are performed by ifetch-misses, they are not useful and the ID filters are able to avoid them.

The increase in the number of data directory panels looked up when using the one-bit ID filter is caused by blocks of 64B that contain both instructions and data. These blocks come from the compiler allocation of constants in the code region. Thus, the first 32B of a 64B block can be instructions and the next 32B can be data. In the system without ID filters, there are copies of the first 32B in the local instruction caches and there are copies of the next 32B in the local data caches. Therefore, both loads and instruction fetches that access this block do not need to access the shared cache. However, in the system with the one-bit ID filter, instruction/data exclusivity at 64B blocks is forced because blocks in the shared cache should be classified as data or instruction. Thus, any ifetch-miss that accesses the first 32B, invalidates all the copies of the next 32B in the local data caches and the other way around. As a result, the number of load-misses and ifetch-misses in the shared cache increases. Both load-misses and ifetch-misses look up directory panels

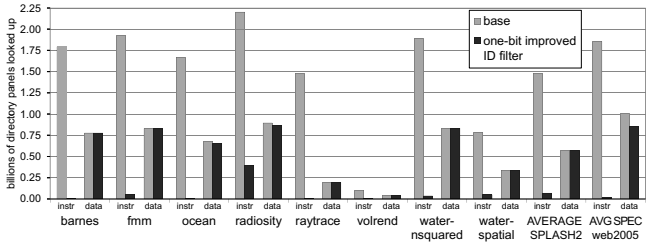

Figure 5: Billions of directory panel lookups when using the one-bit improved ID filter compared with the system without filter. The first two columns correspond to the instruction directory and the next two to the data directory.

(recall Table II). Therefore, there is an increase in the number of directory panels looked up and all of them are useful because there are copies of part of the block in the local caches.

Radiosity is the most affected benchmark because blocks shared by data and instructions are highly accessed. In this benchmark, in order to maintain instruction/data exclusivity at 64B boundary, the number of data and instruction directory panels looked up increases by 26% and 6%, respectively. The one-bit ID filter identifies as useless 8% and 92% of the data and instruction directory panels looked up, respectively. As a result, the number of instruction directory panel lookups is reduced by 92% and the number of data directory panel lookups is increased by 17%.

C. One-bit improved ID filter

When a 64B block contains data and instructions, the number of useful directory panels looked up can increase. This produces an increase in the energy consumed, not only due to the extra directory panels looked up but also due to the increase in the invalidations sent to the local caches.

In order to reduce this waste of energy, we propose a new ID filter. This ID filter, called one-bit improved ID filter, assures instruction/data exclusivity by preventing a block classified as instruction block to change its type. Thus, when a load access a block classified as instruction block, the data is supplied to the local data cache, but it is not allocated in this cache.

In this way, the number of loads in the shared cache increases as with the one-bit ID filter, but neither directory panel lookups nor invalidations are necessary. Moreover, the number of ifetch-misses is the same as in the system without ID filters.

Figure 5 shows the coverage of the one-bit improved ID filter. The first and third columns represent the billions of directory panel lookups performed in the instruction directory and the data directory, respectively. The second and fourth columns show the billions of directory panel lookups when using the one-bit improved ID filter in the instruction directory and the data directory, respectively.

The one-bit improved ID filter identifies as useless 96% of the instruction directory panel lookups for SPLASH2 and 99% for Specweb2005. The number of data directory panel lookups identified as useless decreases to 15% for Specweb2005 and increases to 1% for SPLASH2. It identifies as useless 69% of the total directory panel lookups for SPLASH2 and 64% for Specweb2005.

D. Comparisons avoided per ID filter access

The energy saved by using the ID filters can be estimated by contrasting the number of comparisons avoided with the number of operations performed in the ID filter structure. The two-bit ID filter consists of two bits per block in the shared cache while the one-bit ID filter and the one-bit improved ID filter need only one bit per block. These bits, called from now on filter bits, should be read, written and updated. These operations are the ones we compare with the number of comparisons avoided.

The filter bits are read in every access to the shared cache, written every time that a new block is allocated in the shared cache, and updated when a block already allocated should change its type. In the two-bit ID filter a block can be updated just once from data or instruction to mixed. In the one-bit ID filter, a block can change its type from data to instruction and the other way around as many times as necessary. Finally, in the one-bit improved ID filter, a block can change its type only from data to instruction, so at the most one update per block can be performed.

Figure 4 shows the number of directory panels looked up that can be reduced with each ID filter. Each directory panel lookup action needs to perform a comparison of 32 entries of 15 bits. So, on average, for SPLASH2, using any of the ID filters proposed, we avoid 46 billions of comparisons (1.45*32) of 15 bits during the whole execution of any benchmark. As, on average, we execute 0.85 billions of cycles per benchmark (Table V), 54 out of 77 comparisons per cycle are avoided.

For Specweb2005, on average, the two-bit ID filter avoids 71 billions of comparisons (2.21*32) of 15 bits per workload and either the one-bit ID filter or the one-bit improved ID filter avoids 64 billions of comparisons (2.00*32) of 15 bits per workload. As, on average, we execute 2.25 billions of cycles per workload, the two-bit ID filter avoids 32 out of 41 comparisons per cycle and any of the other two ID filters avoids 29 out of 41 comparisons per cycle.

Figure 6 shows the number of times the filter bits are read, written and updated for each ID filter. Each benchmark has nine columns (many of them are almost 0). The first three corresponds to the two-bit ID filter, the next three corresponds to the one-bit ID filter and the last three to the one-bit improved ID filter. For all the ID filters, the first column represents the number of times that the filter bits are read; the second columns shows the number of times

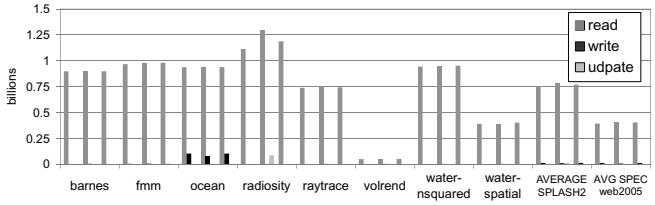

Figure 6: Billions of accesses perfomed to the filter bits in order to read, write or update them

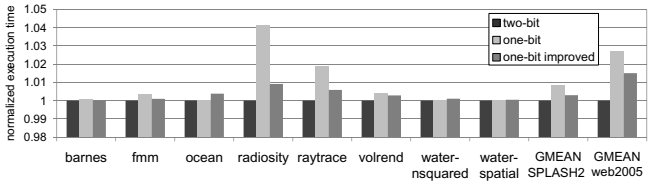

Figure 7: Normalized execution time of the different ID filters proposed with regard to the system without ID filters.

the filter bits are written; and the last column represents the number of times the filter bits are updated.

The number of times that the filter bits are written or updated is negligible compared with the number of times they are read. They are barely written or updated because the shared cache miss rate is low and in general a block does not change its type. The only two exceptions are ocean and radiosity. In ocean, the shared cache miss rate is higher than in the rest and, therefore, the number of times the filter bits are written is 11% of the times they are read. In radiosity, when using the one-bit ID filter, there are several blocks that change its type quite often. In this benchmark, the filter bits are updated 7% of the times they are read.

On average, for SPLASH2, the two filter bits of the two-bit ID filter are read 0.76 billions of times per benchmark, and the filter bit of the one-bit ID filter and one-bit improved ID filter is read 0.78 billions and 0.77 billions respectively. This means that, on average, 0.9 reads are done per cycle. Thus, using any of the proposed ID filters, more than 60 comparisons are avoided for each read to the filter bits.

For Specweb2005, on average, the two filter bits of the two-bit ID filter are read 0.39 billions of times per workload, and the filter bit of the one-bit ID filter and one-bit improved ID filter is read 0.40 billions per workload. This means that, on average, 0.17 reads are done per cycle. Thus, the two-bit ID filter avoids 188 comparisons of 15 bits for each read to the two filter bits and any of the one-bit ID filters avoids 170 comparisons of 15 bits for each read to the filter bit.

E. Performance

The two-bit ID filter does not modify the coherence protocol so benchmarks' performance should not be altered. On the contrary, the one-bit ID filter and the one-bit improved ID filter modify the coherence protocol forcing the instruction/data exclusivity at 64B blocks. It is then necessary to check that the performance of the benchmarks is the same as before.

Figure 7 shows the normalized execution times of the different ID filters proposed with regard to the system without any ID filter. We can see that the performance of the two-bit ID filter is exactly the same as the system without ID filters. Using the one-bit and the one-bit improved ID filters, the performance is slightly worse. The one-bit ID filter is on average 0.8% slower for SPLASH2 and 2.7% slower for Specweb2005, but with the one-bit improved ID filter the system is only 0.2% slower for SPLASH2 and 1.5% for Specweb2005.

Radiosity is the benchmarks with the worst performance due to the blocks that are accessed simultaneously by loads and instruction fetches. Using the one-bit ID filter, radiosity have a performance lost of 4%, but using the one-bit improved ID filter, the performance lost is below 1%.

VI. Related Work

There are several proposals to reduce the power consumption of coherence protocols for both directory-based and snoopy-based protocols. For directory-based protocols, Zebchuk et. al. [17] presents a new directory implementation based on a grid of small bloom filters that uses less area, leakage power and dynamic energy than a conventional directory in a CMP model without a shared cache level. On the other hand, our proposal is intended for a CMP model with a shared cache level and smaller local caches.

For snoopy-based protocols all the proposals are based on the fact that most of the snoop-induced lookups result in a "miss", that is, the snooped caches do not contain the searched block. Depending on the way used to avoid these unnecessary snoop-induced lookups, we can distinguish several groups.

In the first group we classify the proposals that filter the snoop-induced lookups based on the information supplied by a small structure accessed before doing the tag cache lookup. Jetty [6] implements this idea for SMPs. It consists of two small structures per node that respectively represent a subset of not cached blocks and a superset of cached blocks. Ekman et. al. [18] evaluate this proposal in a CMP and determine that for this type of architecture Jetty is not efficient because the power consumed by Jetty is at the level of the power consumed by the local caches. Salapura et. al. [19] implement a structure that keeps a superset of cached blocks. This structure is organized as sets of blocks called Stream Registers. And the Page Sharing Table (PST), proposed by Ekman et. al. [20], uses vectors that identify sharing at the page level. In the last proposal, the information is precise.

The second group of proposals try to avoid not only the snoop-induced lookups but also the broadcast messages.

RegionScout [8] 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, instead of local information about blocks. This reduces the area and energy costs of the filter used in Jetty. Cantin et. al. [9] present a similar idea to RegionScout, but the information kept in the structures is precise and the structures are bigger.

In the third group of proposals for snoop-based protocols, the main idea is to use the knowledge available about the behavior of a program to determine if a region of memory is shared or private and limit snoop-induced lookups to shared blocks. Dash et. al. [21] propose a mechanism in which the compiler reports the shared arrays of an application to the operating system. Ballapuram et. al. [22] propose a hardware that compiler-assisted is able to annotate the different regions that are not shared (stack, global, and heap memory). Any block in these regions is not snooped.

The last group of proposals propose to do the snoops necessary in a more efficient way. Saldanha et. al. [11] propose to serialize snoop messages for load misses in SMPs. The average cache miss latency is increased, but the snoop-related activity is substantially reduced. Ekman et. al. [18] evaluated this idea for CMPs concluding that it does not manage to cut much energy because most of the time the block is not cached in any local cache, so at the end all the local caches are accessed.

VII. Conclusions

We observed that many lookups in the directory of the shared cache were useless because there were no copies of the searched block in any cache in the system. We propose to use an ID filter before accessing the directory which is able to identify in advance whether a lookup is useless or not.

ID filtering keeps track of the memory operations that access each block in an inclusive shared cache in order to be able to classify the block as data or instruction. This is only possible in a system in which the local cache level is split into data and instruction caches. By classifying the block as data or instruction, we know if there are copies of a block only in the local data caches, only in the local instruction caches or in both of them. Thus, the filter can determine if a lookup is useless in either the data directory or the instruction directory and prevent it from being performed.

We propose three different ID filter implementations. All of them use a small area since they only need one or two bits per each block in the shared cache, which represent from 7% to 3% of the tag array of the shared cache (22 bits tag + 6 bits ECC + 4 state bits). Using any of these filters, on average, more than 70% of the directory panels looked up are avoided for SPLASH2 and Specweb2005. To maintain the filters updated, less than a read per cycle of one or two bits are necessary.

In SPLASH2, on average, 1.45 billions of directory panel lookups are avoided per benchmark execution, which represents 46 billions of comparisons of 15 bits, so 54 out of 77 comparisons are avoided per execution cycle. In Specweb2005, on average, more than 2 billions of directory panel lookups are avoided during the simulated time per workload, so more than 64 billions of comparisons of 15 bits are avoided. On average, this means that 29 out of 41 comparisons are avoided per execution cycle.

Acknowledgment

This work was supported in part by grants TIN2007-66423 and TIN2007-60625 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), 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, and M. T. Vaden, "IBM POWER6 microarchitecture," *IBM J. Res. Dev.*, vol. 51, no. 6, pp. 639–662, 2007.

[2] AMD, "AMD Multi-Core Technology," in *http://multicore.amd.com*.

[3] Fujitsu, "Fujitsu SPARC64 VII Processor," June 2008.

[4] Intel, "Leading Virtualization Performance and Energy Efficiency in a Multi-processor Server."

[5] T. Johnson and 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, and 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, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," in *ISCA-15*, May-2 Jun 1988, pp. 280–289.

[8] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," in *ISCA-32*, June 2005, pp. 234–245.

[9] J. Cantin, M. Lipasti, and J. Smith, "Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking," in *ISCA-32*, June 2005, pp. 246–257.

[10] J. Cantin, J. Smith, M. Lipasti, A. Moshovos, and B. Falsafi, "Coarse-Grain Coherence Tracking: RegionScout and Region Coherence Arrays," *Micro, IEEE*, vol. 26, no. 1, pp. 70–79, Jan.-Feb. 2006.

[11] C. Saldanha and M. H. Lipasti, "Power Efficient Cache Coherence," Tech. Rep., 2001.

[12] V. Salapura, M. Blumrich, and A. Gara, "Design and implementation of the blue gene/P snoop filter," in *HPCA-14*, Feb. 2008, pp. 5–14.

[13] *OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification Vol. 1*, Sun Microsystems, Inc., May 2008.

[14] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[15] J. P. Singh, A. Gupta, M. Ohara, E. Torrie, and S. C. Woo, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *ISCA-22*, p. 24, 1995.

[16] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to Simulate 1000 Cores," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 10–19, 2009.

[17] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos, "A Tagless Coherence Directory," in *MICRO-42*, Dec 2009.

[18] M. Ekman, F. Dahlgren, and 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.

[19] V. Salapura, M. Blumrich, and A. Gara, "Improving the Accuracy of Snoop Filtering Using Stream Registers," in *MEDEA '07*, 2007, pp. 25–32.

[20] M. Ekman, P. Stenström, and F. Dahlgren, "TLB and Snoop Energy-Reduction Using Virtual Caches in Low-Power Chip-Multiprocessors," in *ISLPED'02*, 2002, pp. 243–246.

[21] A. Dash and P. Petrov, "Energy-Efficient Cache Coherence for Embedded Multi-Processor Systems through Application-Driven Snoop Filtering," in *DSD '06*, 2006, pp. 79–82.

[22] C. S. Ballapuram, A. Sharif, and H.-H. S. Lee, "Exploiting Access Semantics and Program Behavior to Reduce Snoop Power in Chip Multiprocessors," in *ASPLOS XIII*, 2008, pp. 60–69.