Filtering Directory Lookups in CMPs with Write-Through Caches

Ana Bosque[1], Victor Viñals[2], Pablo Ibañez[2], and Jose Maria Llaberia[1]

[1] DAC, UPC, Barcelona, Spain
{abosque,llaberia}@ac.upc.edu
[2] DIIS, University of Zaragoza, Zaragoza, Spain
{victor,imarin}@unizar.es

Abstract. In CMPs, coherence protocols are used to maintain data coherence among the multiple local caches. In this paper, we focus on CMPs using write-through local caches, and a directory-based coherence protocol implemented as a duplicate of the local cache tags. A large fraction of directory lookups is due to stores performed on private data local to the processor performing the store.

We propose to add a filter before the directory in order to either reduce the associativity of the lookups or even eliminate those that are unnecessary. When a block from the shared cache has only one copy in the local caches, the filter identifies the processor and allows for reducing the number of comparisons performed in the corresponding directory lookup. When that is not possible, the filter bits are used to code other situations that can also reduce the number of directory lookups or their associativity.

We evaluate the fillter in a CMP with 8 in-order processors with 4 threads each and a memory hierarchy with local caches and a shared cache. We show that a filter representing 0.7% of the size of the shared cache can avoid, on average, 97% and 93% of all comparisons performed by directory lookups for SPLASH2 and Specweb2005, respectively. Only for SPLASH2, there is a small performance loss of 0.3%. As a result, on average, directory power is reduced 30.8% and 22.4% for SPLASH2 and Specweb2005, respectively.

1 Introduction

Chip-multiprocessors (CMPs) have become the industry choice of design for high-performance processors. Nowadays, most computer manufactures offer CMPs with different number of cores [20,4,13,16,18], where each of them has at least a local cache level. All CMPs support the shared memory programming paradigm. Thus, local caches need to be kept coherent by means of a coherence protocol.

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. A directory can be implemented in two

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 269–281, 2011.

basic ways: by a full-map [8], or by duplicating the local cache tags [31]. 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.

The number of directory lookups necessary in a coherence protocol depends on the write policy of the local caches. The commercial CMP Niagara 2 [18] uses write-through local caches and a shared cache. It requires more bandwidth than the Piranha prototype [6], which uses write-back local caches, because all stores must access the shared cache. However, the extra bandwidth consumed by Niagara 2 assures that data is always up-to-date in the shared cache. Thus, an access to shared data is serviced directly from the shared cache without any intervention from the local caches. The drawback of write-through local caches, though, is that private stores are sent both to the shared cache and to the directory, where a (probably useless) lookup needs to be necessarily performed. In a CMP like Niagara 2, any store requires a 96-associative directory lookup.

In this paper, we show that many of the directory lookups done by stores are useless. We propose a mechanism to identify stores to private data in order to avoid many lookups in the directory. Furthermore, the mechanism is extended to deal with other situations in which directory lookups can be avoided.

Our results show that for SPLASH2, just by using a filter whose size is 0.7% the size of the shared cache, we can avoid 97% of the comparisons performed inside the directory with a tiny 0.2% performance loss. For Specweb2005, the number of comparisons performed by directory lookups is reduced by 93%. On average, directory power consumption is reduced by 30.8% and 22.4% for SPLASH2 and Specweb2005, respectively.

The rest of this paper is organized as follows. In Section 2, we motivate our work. Section 3 describes the proposed filter. Section 4 shows our experimental results. Section 5 discusses related work and Section 6 contains the conclusions.

2 Motivation

In a directory-based protocol, both stores that access the shared cache and evictions in an inclusive shared cache require a directory lookup in order to invalidate the copies of the block in the local caches.

In a CMP with write-back local caches, stores access the shared cache either on a miss in the local data cache or to get the block ownership and change the coherence state of the block to *Modified*. However, if local caches are write-through, all the stores must access the shared cache. In our workloads (Section 4.2) we found that only 1 out of 100,000 stores access true shared data. Thus, in a CMP with write-through local caches and a duplicate tag directory, when an associative lookup is performed by a store, it happens that most of the times the only copy of the cache block is located in the processor performing the store. The

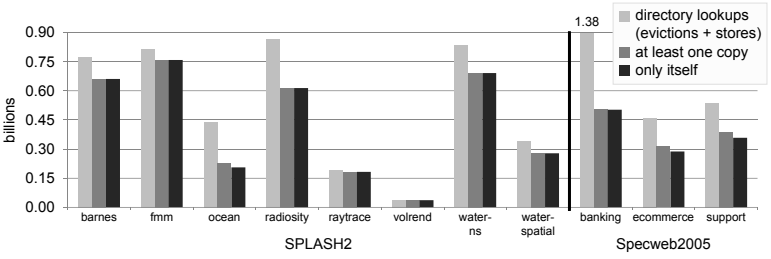


Fig. 1. Billions of directory lookups (`directory lookups`), billions of directory lookups that find at least a copy of the cache block in any local cache (`at least one copy`), and billions of directory lookups performed by stores that only find a copy just in the local cache of the processor that performs the current store (`only itself`).

directory lookups performed by these stores are needless and it is possible to improve directory energy-efficiency by filtering them out.

Figure 1 presents an analysis of directory lookups. Refer to Section 4.1 for the parameters of the simulated CMP model and to Section 4.2 for a description of the workloads used. The difference between the first two bars is the number of times that there are no copies of the shared cache block in any local cache. On average, this difference represents 30% of the directory lookups. The difference between the second and the third bar represents all cases that require to invalidate local cache blocks. These cases are: a) evictions performed over shared cache blocks which have local copies, and b) stores performed over shared cache blocks that are allocated at least in a local cache different from the local cache of the processor performing the store. On average, this difference represents 1% of the directory lookups. The remaining directory lookups (69%) are performed by private stores, i. e. stores that access cache blocks without copies in any other processor's local cache.

The proposed mechanism aims to reduce the number of total directory lookups by filtering out private stores, so that they do not perform expensive and useless directory lookups. Thus, the proposed mechanism reduces the number of directory lookups shown in the third column in Figure 1. Additionally, the filter is enhanced in order to also avoid the 30% of directory lookups that do not find any copy in the local caches, and that are also useless. In Figure 1, these directory lookups are represented by the difference between the first and the second column.

3 Filtering Mechanism

3.1 Overview

We assume a CMP with a shared inclusive L2 cache and multithreaded processors that access local instruction and write-through data caches. A detailed description of the CMP model is in Section 4.1.

As processors are multithreaded, local caches are highly accessed by the processors. A directory organization such as a full-map directory requires a lookup in the local cache tags for every invalidation. As a result, if processor requests and invalidations sent from the directory share the same local cache port, thread execution can be delayed. Thus, the local cache tags require two ports so that thread performance is not diminished. An alternative is to replicate the cache tags [28,9]. This replica is located side-to-side with the local cache tags and it is used by invalidations to set the state bits of the cached blocks.

The replica of the local cache tags can be located in the other side of the interconnection network and be used as a duplicate tag directory. The full-map directory is removed. Now, when an invalidation is sent, the local cache set and way to invalidate is already identified in the message, and a local cache lookup is not needed. As the replica of the local cache tags is located together with the inclusive shared cache, it is possible to keep pointers to the shared cache tags (set index and way) instead of the local cache tags themselves. Consequently, the duplicate tag structure is much smaller [18].

Every directory lookup requires an expensive associative lookup in the duplicate tag structure. However, using a full-map directory, only the tags of the processors effectively having a copy of the block are looked up. Based on program behavior, we propose to use a filter before accessing the duplicate tag directory in order to reduce the lookup associativity. Figure 1 shows that, on average, 69% stores are private. If we identify these cases, the lookup in the duplicate tag directory can be restricted to the duplicate tag of the processor performing the private store (in order to determine which cache way to update in the local cache, see Section 4.1).

The proposed filter manages the same information than a DIR_1NB directory scheme [1], but it is only used as a filter before looking up in the duplicate tag directory. In a DIR_1NB directory, the only processor than can have the copy of a block (owner) is identified. Thus, the proposed filter has as many entries as lines in the shared cache, and each entry has $\log_2 P$ bits plus a valid bit. When the valid bit is zero, the representation of the owner identifier bits is changed to a coarse granularity [14,19]. Consequently, other situations might be identified, for example, whether there are no copies of a block in any local cache. Figure 1 shows that 30% directory lookups are performed under these conditions.

3.2 Filter Operation

Each line in the shared cache has associated one entry in the filter. For every shared cache access or eviction (memory operation from now on), the filter entry is read together with the state bits of the line. Depending on the value stored in the filter, the directory lookup performed by any memory operation accessing that line can be either eliminated or performed over a smaller number of entries in the directory structure.

A filter entry state is updated using only the following information: memory operation type, identifier of the processor performing the memory operation, and previous filter state. We also know the evictions from local caches. Using them

Table 1. Filter states

valid bit	owner identifier	information	filter state
1	xxx	xxx is the only processor that can have a local copy of the block in its local data cache	valid owner
0	000	there are no copies of the block	no copies
0	001	block cached only in the local data caches of processors identified as 0xx	data block (subgroup0)
0	010	block cached only in the local data caches of processors identified as 1xx	data block (subgroup1)
0	011	data block	data block (all)
0	100	unused	
0	101	block cached only in the local instruction caches of processors identified as 0xx	instruction block (subgroup0)
0	110	block cached only in the local instruction caches of processors identified as 1xx	instruction block (subgroup1)
0	111	instruction block	instruction block (all)

the filter information will be precise, but extra directory accesses and costly filter updates will be required. Consequently, we decide to not keep filter information precise all the time, that is, to only know a superset of the copies in the local caches.

3.3 Filter States

The modeled CMP has 8 cores, so a filter entry has 3 owner identifier bits and a valid bit. Table 1 shows how these bits are used to encode different filter states that will reduce directory lookups or directory lookups associativity.

The directory is split in data and instruction directories. Most directory lookups are performed on both directories (only lookups performed to keep instruction/data exclusivity are performed in only one directory (see Section 4.1)). As long as the filter identifies the type of the block (*data or instruction block*), directory lookups are limited to just one directory. Moreover, if the owner (*valid owner*) or the owner's group (*subgroupX*) is identified, the lookup associativity is reduced since only the entries of the owner or its group have to be looked up. Finally, directory lookups are completely avoided if the filter indicates that there are no copies of the block in any local cache (*no copies*).

The filter state *valid owner* is set on three cases: a) local data cache misses that also misses in the shared cache, b) local data cache misses to a block in the shared cache without copies in the local caches (*no copies*), and c) store to a block in the shared cache which may have copies in the local data cache of the processor performing the store (*valid owner* equal to the processor performing the store, *data block (all)*, or *data block (subgroupX)* where 'X' is the subgroup which the processor performing the store belongs to).

The filter state is set to *no copies* in two cases: a) store that misses in the shared cache, and b) store to a cache block in the shared cache which is not present in the local data cache of the processor performing the store (*no copies*, *valid owner* when the owner is different from the processor performing the store, *instruction block*, or *data block (subgroupX)* where 'X' is not the subgroup the processor performing the store belongs to).

Both local instruction and data cache misses modify the filter state to add the processor performing them as one of the processors that can have a copy of the accessed block in its local caches. When a local data cache miss accesses a block whose filter state is *instruction block*, the filter state is not modified.

3.4 Filter Overhead

The filter proposed requires $(1 + \log_2 P)$ bits per shared cache line, being P the number of cores in the CMP. For the CMP described in Section 4.1, as it has 8 cores, four extra bits per shared cache line are required. For each 512KB, 64B block-size L2 bank, the filter implementation requires 4KB. This represents 12% of the tag array size (including the state bits in the tag array) and 0.7% of the total bank size (tag array + data array).

4 Evaluation

4.1 Chip Multiprocessor Model

Figure 2 shows the CMP configuration we assume in this work. It is a CMP with 8 in-order multithreaded cores with 4 threads each and a memory hierarchy similar to the one in Niagara 2 [18]. 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 also has a store buffer (SB) with several entries per thread that contain all outstanding stores. The second-level cache (L2), which is inclusive, is shared among all the cores. It is divided into 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 the 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 2 collects the specific parameters we chose for the memory hierarchy.

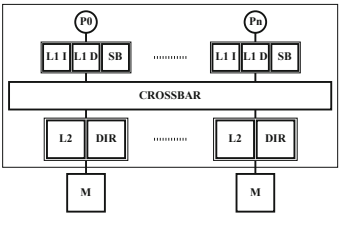


Fig. 2. CMP model

Table 2. Memory hierarchy parameters

L1 D size	8KB	L2 size	4MB
L1 D assoc.	4-way	L2 no. banks	8
L1 D block	16B	L2 assoc.	16-way
L1 I size	16KB	L2 block	64B
L1 I assoc.	8-way	L2 latency	7 cycles
L1 I block	32B	L2 MSHR	8
store buffer	8 entries per thread	Crossbar arb.	3 cycles
		Crossbar lat.	3 cycles
Phys. address	40 bits	Memory lat.	117 cycles

We assume a directory similar to that of Niagara 2 [30], 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 gives

Table 3. SPLASH2 benchmarks.

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-ns	2192 particles	13.79	1.72
water-spatial	4096 particles	4.02	0.50

Table 4. Specweb2005 workloads.

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

the way or ways of the local caches where the copies of the subblocks are located. Thus, an invalidation message consists of the local cache set and way to invalidate. Stores update local caches when the ack message is received. The ack message includes the way where the copy of the block is located in order to avoid the local cache lookup.

Like in Niagara 2 [30], 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). The proposed filter has only one entry for every shared cache line. As a result, instruction/data block exclusivity has to be maintained at a shared cache block size granularity to guarantee the correct filter operation.

4.2 Methodology

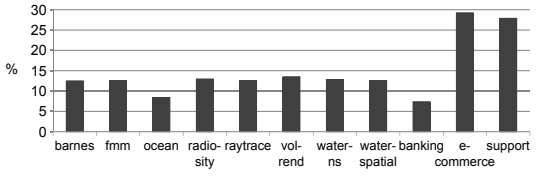
We use a Simics-based simulator. Simics [21] is a full-system multiprocessor simulator capable of running unmodified commercial OSs and applications. We configured Simics to model a SPARC V9 target system with a Total Store Order (TSO) consistency memory model running Solaris 9.

We use the applications from the SPLASH2 benchmark suite [27] and, as non-numerical applications, the three workloads from Specweb2005: Banking, Ecommerce, and Support [15]. In order to adapt the SPLASH2 workloads to our simulated scenario, we scaled the input dataset up as proposed by Monchiero et al. [22] (Table 3). Due to simulation time restrictions, we cannot simulate as many Simics processors in Specweb2005 as in SPLASH2. As a result, Specweb2005 applications are executed in a CMP with 8 non-multithreaded processors.

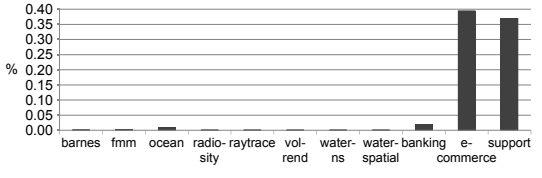
For the three workloads of Specweb2005, we use Apache 2.0.63 web server. Web servers present high time and space variability [3] (Table 4). Conclusions are based on the mean of the simulations and on statistical techniques used by Alameldeen et. al. [3]. To determine the number of web transactions in 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. In the rest of the paper, for Specweb2005 results we show the mean of all simulation runs.

4.3 Filter Coverage

Figure 3 shows the percentage of comparisons performed by the directory lookups in the CMP with the proposed filter with respect to the comparisons performed without filtering. Table 5 shows the number of comparisons performed in the system without filtering.



(a) data directory



(b) instr directory

Fig. 3. Percentage of directory comparisons performed by directory lookups when filtering

Table 5. Billions of directory comparisons without filtering

benchmark	data dir	instr dir
barnes	24.87	57.34
fmm	26.54	61.56
ocean	21.73	53.38
radiosity	28.61	70.45
raytrace	6.23	47.29
volrend	1.28	3.23
water-ns	26.70	60.41
water-spatial	10.82	25.03
banking	45.21	90.09
ecommerce	24.84	43.87
support	28.33	48.59

Figure 3 shows that the number of comparisons performed by directory lookups is reduced, on average, by 93%. The reduction is more important in the instruction directory: more than 99% in the instruction directory for all the benchmarks vs. 88% for SPLASH2 and 81% for Specweb2005 in the data directory.

A data directory lookup is necessary to determine if a block in the local data cache has to be updated and the way in which the copy of the block is located. For this reason, there are several comparisons in the data directory that can not be eliminated.

For Ecommerce and Support, the reduction in data directory comparisons is lower than in the other benchmarks. More than 10% of their stores access cache blocks that are in *data block* filter state, while in the other benchmarks this number is below 1%. That means that the amount of shared data is also larger than in the rest of benchmarks. In this situation, the number of needless directory lookups is smaller, and so the number of comparisons to avoid.

4.4 Performance

The proposed filter modifies the coherence protocol forcing the instruction/data exclusivity at a 64B granularity. Thus, we need to check that the performance

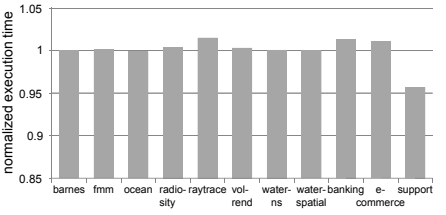


Fig. 4. Normalized execution time

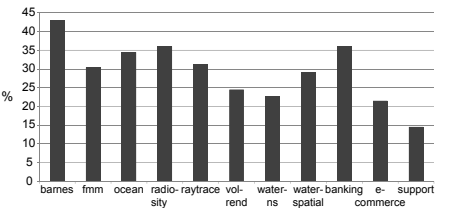


Fig. 5. Percentage of power reduction in the directory

remains unchanged. Figure 4 shows the normalized execution time of the CMP with the filter proposed with respect to the baseline CMP. In SPLASH2 all benchmarks show a performance loss below 0.5%, except raytrace, which has a performance loss of 1.5% due to the increase in the local data cache miss rate. In Specweb2005 we can differentiate two groups: for banking and ecommerce, the mean execution time shows an increase of 1.4% and 1.1%, respectively, in support, the mean shows an execution time decrease of 4.3%. In both groups the confidence interval shows that the execution time is not statistically different.

4.5 Power Consumption

CACTI 6.5 [25] is used to estimate dynamic energy and leakage power for the shared cache tag array and the proposed filter. We modified CACTI to model CAM structures, so that the directory energy consumption, both dynamic and static, can be estimated. All structures were modeled using a 65nm technology with a target frequency of 1.2GHz.

The average dynamic power consumption is computed based on activity statistics of the shared cache, the filter, and the data and instruction directories collected during benchmark execution. The average dynamic power consumption of the directory is 1.5 times the average dynamic power consumed by the tags of the shared cache. However, the leakage power of the tags is 2.2 times the directories leakage since these structures are smaller than the tags of the local caches.

Figure 5 shows the percentage of power reduction in the directory using the filter proposed. It takes into account power reduction in the directory as well as additional consumption due to the filter structure embedded into the shared cache tags. The directory power consumption includes the dynamic power and the leakage power in both data and instruction directories. The proposed filter is placed together with the shared cache tags, so tags and filter state bits are read together in every access to the shared cache. This means that both the energy consumed by the shared cache tag array on any operation and its leakage power increase. These increases affect the energy reduction in the directory. The energy to update the filter state also decreases the dynamic energy reduction in the directory.

On average, the directory power is reduced by 30.8% for SPLASH2 and by 22.42% for Specweb2005. The difference between SPLASH2 and Specweb2005 is due to simulating Specweb2005 in single-thread processors. Ecommerce and Support show a shared cache miss rate higher than the rest of benchmarks. As there is only 1 thread per core, every shared cache miss stalls a core and, as a result, the number of accesses to the shared cache and directory lookups are smaller than in the rest of benchmarks. Thus, the dynamic power reduction in the directory is smaller, 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.

Other cache configurations and new generation technologies. The size of the proposed filter is directly proportional to the number of shared cache lines. Moreover, the energy consumed by the directory depends on the number of directory lookups performed which is determined by the shared cache accesses. If the size of the local caches is increased, the shared cache accesses are modified. Thus, we decide to analyze the reduction of power for different cache configurations. We simulate a CMP in which the sizes of the shared cache and the local caches are doubled. The percentage of power reduction is smaller than in the baseline system due to the increase in the power consumption of the proposed filter (bigger shared cache) and the decrease in the number of directory lookups performed (bigger local caches). On average, in the worst case, the percentage of power reduction is 24% for SPLASH2 and 10% for Specweb2005.

Finally, we analyze how the percentage of power reduction is affected for new generation technologies. We model all structures using a 22nm technology with a target frequency of 2.75GHz. On average, the percentage of power reduction is 19.5% for SPLASH2 and 10.5% for Specweb2005.

5 Related Work

This section gathers together several techniques to filter out coherence actions, e.g., local cache lookups or broadcast messages. The filter is either placed together with the local caches or distributed in 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 [24] adds small structures to SMPs that are accessed before doing the tag cache lookup and Ekman et al. [11] evaluate this proposal on CMPs. Salapura et al. [26] propose a structure that keeps a superset of cached blocks. The Page Sharing Table (PST), proposed by Ekman et al. [12], uses vectors that identify sharing at the page level with precise information.

There is a group of proposals that try to not only filter snoop-induced lookups but to reduce broadcast messages. RegionScout [23] implements several structures per node in a similar way to JETTY [24], but these structures keep global

system information about regions, which are continuous sections of memory. Cantin et al. [7] 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. [29] 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 behaviour of a program helps determining whether a region of memory is shared or private and limit snoop-induced lookups to shared blocks [10,5].

There are proposals that distribute the filter over the on-chip network for snoopy-based and directory-based protocols. Agarwal et. al. [2] 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. Jerger [17], in a coarse-grain like directory-based protocol, adds counting bloom filters 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.

Unlike previous proposals, the goal of the proposed filter is to reduce energy consumption in a coherence directory implemented as a duplicate tag directory in a CMP with write-through caches. This CMP is similar to Niagara 2 that has a limited number of cores. However, if the number of cores in the system increases significantly (many-cores), cores could be organized in groups or clusters. Every cluster might work like a small CMP with write-through local caches since the coherence protocol inside the cluster is greatly simplified. The shared cache in a cluster would be private for that cluster. It could use a write-back policy to update the last-level cache shared among all the clusters or main memory. Our filtering mechanism would be used inside each cluster. However, such systems are out of the scope of this paper.

6 Conclusions

We have observed that in CMPs with write-through caches, a big fraction of directory lookups is due to stores performed over data that are private to the processor executing the store instruction. In such a situation, a directory lookup is performed but no invalidations are necessary. This needless directory lookup wastes energy. We propose to use a filter before accessing the directory. The filter is able to identify private stores and reduce the number of directory lookups performed or the number of directory entries looked up in a directory lookup.

The proposed filter has an entry for each line in the shared cache. For every shared cache access, a filter entry is read together with the state bits of the block accessed. Every filter entry keeps either the owner of the corresponding block or some useful information to limit the associativity of a directory lookup

performed over the corresponding block. Using this information the number of comparisons in the directory is greatly reduced.

The proposed filter area is 12% the tag array area and 0.7% the total shared cache area, and filtering is performed on every access to the shared cache. Our results show that, on average, the proposed filter reduces the number of comparisons performed by directory lookups by 95%, and reduces the directory power by 28.2% for all the benchmarks.

References

[1] Agarwal, A., Simoni, R., Hennessy, J., Horowitz, M.: An Evaluation of Directory Schemes for Cache Coherence. In: ISCA-15, pp. 280–289 (1988)
[2] Agarwal, N., Peh, L.-S., Jha, N.: In-Network Coherence Filtering: Snoopy coherence without broadcasts, pp. 232–243 (2009)
[3] Alameldeen, A.R., Wood, D.A.: Variability in Architectural Simulations of Multi-Threaded Workloads. In: HPCA-9, p. 7 (2003)
[4] AMD. AMD Multi-Core Technology, `http://multicore.amd.com`
[5] Ballapuram, C.S., Sharif, A., Lee, H.-H.S.: Exploiting Access Semantics and Program Behavior to Reduce Snoop Power in Chip Multiprocessors. In: ASPLOS XIII, pp. 60–69 (2008)
[6] Barroso, L.A., et al.: Piranha: a Scalable Architecture Based on Single-Chip Multiprocessing. In: ISCA-27, pp. 282–293 (2000)
[7] Cantin, J.F., Lipasti, M.H., Smith, J.E.: Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In: ISCA-32, pp. 246–257 (June 2005)
[8] Censier, L.M., Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems. IEEE Transactions on Computers C-27(12), 1112–1118 (1978)
[9] Charlesworth, A., Aneshansley, N., Haakmeester, M., Drogichen, D., Gilbert, G., Williams, R., Phelps, A.: The Starfire SMP Interconnect, p. 37 (1997)
[10] Dash, A., Petrov, P.: Energy-Efficient Cache Coherence for Embedded Multi-Processor Systems through Application-Driven Snoop Filtering. In: DSD 2006, pp. 79–82 (2006)
[11] Ekman, M., Dahlgren, F., Stenström, P.: Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors. In: Workshop on Duplicating, Deconstructing and Debunking, in conjunction with ISCA (May 2002)
[12] Ekman, M., Stenström, P., Dahlgren, F.: TLB and Snoop Energy-Reduction Using Virtual Caches in Low-Power Chip-Multiprocessors. In: ISLPED 2002, pp. 243–246 (2002)
[13] Fujitsu. Fujitsu SPARC64 VII Processor (June 2008)
[14] Gupta, A., dietrich Weber, W., Mowry, T.: Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In: ICPP 1990, pp. 312–321 (1990)
[15] `http://www.spec.org/web2005/`
[16] Intel. Leading Virtualization Performance and Energy Efficiency in a Multiprocessor Server
[17] Jerger, N.: SigNet: Network-on-chip filtering for coarse vector directories. pp. 1378–1383 (2010)
[18] Johnson, T., Nawathe, U.: An 8-core, 64-thread, 64-bit Power Efficient SPARC SOC (niagara2). In: ISPD 2007, p. 2 (2007)

[19] Laudon, J., Lenoski, D.: The SGI Origin: A ccnuma Highly Scalable Server, pp. 241–251 (1997)
[20] Le, H.Q., et al.: IBM POWER6 microarchitecture. IBM J. Res. Dev. 51(6), 639–662 (2007)
[21] Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer 35(2), 50–58 (2002)
[22] Monchiero, M., Ahn, J.H., Falcón, A., Ortega, D., Faraboschi, P.: How to Simulate 1000 Cores. SIGARCH Comput. Archit. News 37(2), 10–19 (2009)
[23] Moshovos, A.: RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In: ISCA-32, pp. 234–245 (June 2005)
[24] Moshovos, A., Memik, G., Falsafi, B., Choudhary, A.: JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In: HPCA-7, 2001, pp. 85–96 (2001)
[25] Muralimanohar, N., Balasubramonian, R.: CACTI 6.0: A Tool to Model Large Caches (2009)
[26] Salapura, V., Blumrich, M., Gara, A.: Improving the Accuracy of Snoop Filtering Using Stream Registers. In: MEDEA 2007, pp. 25–32 (2007)
[27] Singh, J.P., Gupta, A., Ohara, M., Torrie, E., Woo, S.C.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: ISCA-22, p. 24 (1995)
[28] Steinman, M.B., Harris, G.J., Kocev, A., Lamere, V.C., Pannell, R.D.: The AlphaServer 4100 Cached Processor Module Architecture and Design (1996)
[29] Strauss, K., Shen, X., Torrellas, J.: Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. SIGARCH Comput. Archit. News 34(2), 327–338 (2006)
[30] Sun Microsystems, Inc. OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification vol. 1 (May 2008)
[31] Tang, C.K.: Cache System Design in the Tightly Coupled Multiprocessor System. In: AFIPS 1976, pp. 749–753 (1976)